# CONTRIBUTIONS TO THE MODELLING AND THE USE OF SOFTWARE PRODUCT LINES

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea "Politehnica" din Timişoara, România
şi
L'Université de Nice Sophia-Antipolis, France
în domeniul ŞTIINŢA CALCULATOARELOR
de către

**Ing. Emanuel Ţundrea**

| | |
|---|---|
| Conducători ştiinţifici: | prof.univ.dr.ing. Ioan Jurca |
| | prof.univ.dr. Philippe Lahire |
| Referenţi ştiinţifici: | prof.univ.dr.ing. Vladimir-Ioan Creţu |
| | prof.univ.dr.ing. Ioan Salomie |
| | conf.univ.dr.ing. Alexandru Cicortaş |

Ziua susţinerii tezei: 09.01.2009

România, 300159 Timişoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

# Acknowledgments

Every time I think of the following friends my heart is filled with gratitude. Here are some well deserved pats on the back:

*Philippe Lahire*, you deserve heart-felt applause and an early retirement. Thank you for those one million and one acts of kindness.

*Ioan Jurca*, you reached the highest standards of teaching quality and integrity! You also write so skilfully in English: thank you for your help which made others think that I write well :)

*Pierre Crescenzo,* knowing that you had to assist such a slow-witted author, it is obvious that you should get the credit for writing this thesis. I am so thankful.

*Dan Pescaru, Ciprian Chirila and Emanuel Sasu,* Amigos queridos! It is a privilege for me to be part of your team!

*John Lenton*, your encouragement and support have been priceless. This would not have been possible without your help. Your class on 'time management' changed my life!

*Mark and Gill Mumford,* how gifted you are and what a gift to me you have been! If there is another more hospitable home, I can witness that I did not find it.

*My Father and my Mother,* one of the best men on earth and probably one of the best women in heaven. Your devotion to me was surpassed only by your devotion to Christ.

*My Wife, Nadia,* if a star fell every time I thought of you during these years, the sky would be empty. Our marriage proved to me that heaven is real and it is not necessary to die to arrive there.

*My kids, Natanael and Titus,* the boys who lift my heart. There is nothing I am more proud about.

And more than anybody, *to You, my GOD*: the only reason I can put down a burden is because you are there to carry it for me. Because of Calvary I am free to choose and so I choose to honour you the most.

Timişoara,
January 2009

Eng. Emanuel Ţundrea

**Ţundrea, Emanuel**

**Contributions to the Modelling and the Use of the Software Product Lines**

Teze de doctorat ale UPT, Seria 10, Nr. 11, Editura Politehnica, 2009, 152 pagini, 48 figuri, 6 tabele.

ISSN: 1842-7707

ISBN: 978-973-625-793-3

Cuvinte cheie: model, software product lines, aspect oriented programming, generative programming, prototype, software factory

Rezumat,
In software engineering everything evolves very fast: user requirements, technologies, methodologies and applications. Can we foresight and strengthen our approaches to build software to confront these more and more complex challenges? While there are key issues to solve, it is also noteworthy to know that we are very close to exciting innovations. Software Product Lines (SPL) - modeling technology together with source-code generative tools seem to make it easier to manage diverse environments with complex. constantly changing relationships.

In the context of SPL, this thesis promotes the idea that the most promising way to address software engineering is *to provide an approach centred on models which captures the know-how of a domain, independently from both the software platform and the possible applications*.

In this context, this thesis concentrates on providing a solution to respond to two main questions:

- *How to develop model-oriented software?* In other words, it presents SmartModels approach which offers a flexible and easy way to describe the know-how of a business domain into models independent from the technology. Technology evolves continuously in all aspects - platforms, programming languages, etc., and as a result it is important for companies to find a solution to avoid investing in placing their know-how directly at the level of a software platform, and to invest to a meta-level instead (using parameterized genericity).
- *How to build software product lines?* In other words, SmartModels provides a solution for describing concepts - a new method of abstracting the domain entities into the model level, methods to handle the commonalities, but also the differences between products, which in addition call for the need to impose constraints. As a result a company gains in terms of productivity and adaptability. It also fosters new opportunities for easier evolution of the semantics of the domain entities and for better reusability which again results in increasing companies' productivity.

Therefore, the thesis aims to take part in the modelling and the use of software product lines.

## Table of Contents

## Table of Figures

## List of Tables

## List of Abbreviations

| | |
|---|---|
| ACM | Association for Computing Machinery |
| ANSI/SPARC | American National Standards Institute / Standards Planning And Requirements Committee |
| AOP | Aspect Oriented Programming |
| AST | Abstract Syntax Tree |
| BNF | Backus Naur Form |
| CAFÉ | from Concepts to Application in system-Family Engineering |
| CCM | CORBA Component Model |
| CLOS | Common LISP Object System |
| COBOL | COmmon Business-Oriented Language |
| CORBA | Common Object Request Broker Architecture |
| CVS | Concurrent Versions System |
| DDD | Domain Driven Development |
| DSLs | Domain-Specific Languages |
| DTD | Document Type Definition |
| EDOC | Enterprise Distributed Object Computing |
| e.g. | exempli gratia |
| EJB | Enterprise Java Beans |
| EMF | Eclipse Modelling Framework |
| ESAPS | Engineering Software Architectures Processes and Platforms for System-Families |
| ESI | European Software Institute |
| FAMILIES | FAct-based Maturity through Institutionalisation. Lessons-learned and Involved Exploration of System-family engineering |
| GEF | Graphical Editing Framework |
| GMT | Generic Mapping Tools |
| GUI | Graphical User Interface |
| IBM | International Business Machines |
| i.d. | id est |
| IDE | Integrated Development Environment |
| IDL | Interface Definition Language |
| J2EE | Java 2 Platform, Enterprise Edition |
| J2SE | Java 2 Platform, Standard Edition |
| JDT | Java Development Tools |
| JET | Java Emitter Templates |
| JSP | JavaServer Pages |
| IP | Intentional Programming |
| LAN | Local Area Network |
| LOOPS | LISP Object-Oriented Programming System |
| MDA | Model Driven Architecture |
| MDE | Model Driven Engineering |
| MOF | Meta Object Facility |
| MOP | Model Oriented Programming |
| *mop* | Meta-Object Protocol |
| OA&D | Object Analysis and Design |

| | |
|---|---|
| OFL | Open Flexible Language |
| OMG | Object Management Group |
| OMT | Object Modelling Technique |
| OOA | Object Oriented Analysis |
| OOM | Object-Oriented Modelling |
| OOD | Object-Oriented Design |
| OOPL | Object-Oriented Programming Language |
| OOSE | Object-Oriented Software Engineering |
| PDE | Plug-in Development Environment |
| PFE | Product Family Engineering |
| RCP | Rich Client Platform |
| SDK | Software Development Kit |
| SDO | Service Data Objects |
| SEI | Software Engineering Institute |
| SOA | Service-Oriented Architecture |
| SOP | Subject-Oriented Programming |
| SPL | Software Product Lines |
| SPLC | Software Product Line Conference |
| UML | Unified Modelling Language |
| W3C | World Wide Web Consortium |
| XML | eXtensible Markup Language |

# 1. Introduction

The impact of technology on all aspects of contemporary life is an unchallenged fact. We alternately suffer with and revel in the side effects of technology: from toxic waste, the social effects of too much television, and carpal tunnel syndrome, to the wonders of spreadsheets or tiny video cameras. Yet technology effects, both good and bad, seem always to take us by surprise because it is so fast evolving that we hardly can anticipate its course.

The computer software industry is a large and rapidly growing industry throughout the industrial world. It produces an enabling technology that affects manufacturing firms' commercialization of new products in a growing array of industries. The availability of software with greater flexibility and ease of use also influences the pace of adoption of information technologies and thereby affects the realization of the productivity gains associated with this technology.

In considering productivity issues vs. staying on track with the new technologies there is also another theme which clearly emerges: the joys and pains of managing evolution and, in particular, the changes in computer-science entities. The Internet, broadband, multimedia, switched LANs, electronic commerce, groupware, wireless and a host of other innovations are coming at us with frightening swiftness and all our software must support them in order to put on competitive advantages on the market. The Internet is also speeding the adoption of a diverse set of industrial-strength, encryption-based security technologies, which will present their own management challenges.

Can we foresee and strengthen our approaches to build software to confront these challenging situations? While there are major challenges in management systems, it is also important to note that we are on the verge of exciting innovations. For example, modelling technology plus source-code generative tools make it easier to manage diverse environments with complex, constantly changing relationships.

The more and more rapid technology development and assessment of new platforms both hardware and software, corroborated with the fast increase of the complexity of the user applications requirements lead to new paradigms in software engineering which meet two important conditions: *shorter time for deploying* a solution to the market and *better modelling its architecture* so it can evolve, be adapted and extended easier.

It is also important to discuss how the architects can reduce the crucial gap between modelling methodologies and programming languages, like object-oriented ones, which do not offer native support for high level abstractions and notations. Modelling methodologies are an important step forward and this thesis concentrates and tries to benefit from and contribute to the current research in this domain.

## 1.1. Motivation: Paradigm Shift in Software Engineering

"*Objects everywhere!*" Starting in the late 1960s [37] with programming languages such as Simula, then Smalltalk, objects have pervaded every domain of software technology. It is true that objects have proliferated like a contagious virus.

We all thought that because of the wonderful unifying properties of the object paradigm, the transition from procedural technology to object technology will bring huge conceptual simplification to the software engineering field. And it surely did at that time!

Since everything will be considered as an object, we shall observe a dramatic reduction in the number of necessary concepts. But as we can see today, the more and more complex software we need to develop and more and more complex technologies we need to deal with, object technology has failed to really achieve its promises of simplification. There is still considerable confusion and controversy over such key concepts as encapsulation, inheritance and polymorphism. There is also a lack of a complete theory of object orientation, based on simple and well-defined concepts [74]. Some theories to aid understanding of objects have been proposed, but they are incomplete, concentrating on a few aspects such as polymorphism [28] [40] [100].

The reality is that the object-oriented approach does not provide all the solutions for software engineering even if (and this is important to bear in mind from now on) it represents a valuable basis for the description of further approaches. This remark can also be applied to *component-based software engineering* [100] and to the newer *web services* paradigm [46]. In particular, they do not provide a correct answer to the continuous evolution of the technologies: keeping applications up-to-date according to the evolution of technologies is too much time-consuming.

In the past two decades, there were many changes in computer science that had an influence upon the way an application can be developed. To cope with these changes, applications need to be more open, adaptable and ready for fast evolution. Before going any further, here are some arguments why these new constraints in software development have emerged [102].

The first reason is the emergence of Internet which implied applications are no longer stand-alone, but rather distributed. Therefore, from now on data communication between applications and users must be taken into account during the whole application life-cycle. One important point is to choose a well adapted data exchange format.

The second reason of these changes is the proliferation of new component technologies. It is difficult to choose the right and more capable among them. For instance, for a component based application, a developer must choose between, at least, three component technologies: CCM (CORBA Component Model), EJB (Enterprise Java Bean), or Web Services.

The third reason is the democratization of computer science. Users may have now different knowledge levels, different needs, a wide range of visualization devices, and specific activity domains. This aspect should be considered when designing and developing applications.

The last reason is business related. Indeed, to be efficient and competitive a company must quickly and cheaply adapt its software to new user needs and technologies. Time to market of a software product is smaller and companies face a huge pressure on this aspect, but this factor should not shortcircuit the absolute need for quality.

The above paragraphs have already justified that object-oriented programming is not always sufficient to handle clear designs and reusable developments of software. For example, concerns can be cross-cut between classes and there can be a mix between functional and non-functional code in a single class making the code difficult to maintain and debug. This situation explains the emergence of a whole set of new programming paradigms such as Aspect Oriented Programming (AOP) [66], Subject Oriented Programming (SOP) [48], Intentional Programming (IP) [94], or Component Programming [100] [49].

At the specification level, a strong and continuous work is undergoing toward standards of the World Wide Web Consortium (W3C) [46] for documents or of the Object Management Group (OMG) for design methodologies such as Unified Modelling Language (UML) [80] or Model Driven Architecture (MDA) approach [86].

This is the motivation for which today, the world of software engineering arrived to a paradigm shift from *object technology* to *model technology*, from *object composition* to *model transformation*. From objects and components, we can see other evolving trends like: processes, rules, services. Model technology is able to subsume most of these paradigms and others [59].

Therefore, *"model once, generate everywhere"* is the assertion that put a mark on the paradigm shift generated by the MDA, but the road to model engineering takes time.

I close this section with the quote which I found to be very keen to our problem: *"the good thing about bubbles and arrows, as opposed to programs, is that they never crash"* [73]. Under these circumstances, section 1.2 will introduce the approach proposed by this thesis.

## 1.2. Approach Proposal

Based on the motivation developed in section 1.1, this thesis promotes the idea that the most promising way to address software engineering is to provide an approach centred on models which captures the know-how of a domain, independently from both the software platform and the possible applications.

The effort will be spent on researching two tracks (both for theorizing an approach to answer these questions and for building a prototype to validate it):

*How to develop model-oriented software?* (see section 2.3) In other words, it is important to offer an easy way to describe the know-how of a business domain into models independent from the technology. This follows the paths opened by Model Driven Engineering (MDE) approaches, from which the best known initiative is Model Driven Architecture (MDA) [86] from Object Management Group (OMG) [45]. Technology evolves continuously in all aspects – platforms, programming languages. Therefore, it is important for companies to find a solution (see chapter 3) to avoid investing in placing their know-how directly at the level of a software platform, and to invest to a meta-level instead. This is the goal of MDE approach: to insure that the know-how is encapsulated in models, and, as a result, it is technology independent.

*How to build software product lines?* (see section 2.4) In other words, it is important to provide a solution for describing concepts – a new method of abstracting the domain entities into the model level. Most of the models require the description of more than just only one root concept and they have commonalities (share same characteristics), but also differences which in addition call for the need to impose constraints. Software Product Lines (SPL) [76] approaches aim to answer and offer support for building lines of products by providing techniques to write their semantics. As a result a company gains in terms of productivity and adaptability. In this context, besides the classic but important inheritance mechanism taken over from object oriented approach, the variability of products is also modelled through the notion of genericity. This is nurturing new opportunities for easier evolution of the semantics of the domain entities and for better reusability which again results in increasing companies' productivity (see chapter 4).

Therefore, this thesis proposes an approach which aims to enhance productivity by:

- handling the know-how of a domain in a platform independent manner;
- improving the expressiveness of models by introducing the notion of family of entities as a parameterized concept (integrating customizable genericity);
- increasing the reusability by making the contents of models more adaptable.

I expect that this approach will inspire a solution which fills the gaps between the modelling solutions used by architects, software quality that engineers hope for, quantity of source-code that programmers have to write, and productivity targets that companies have to reach. Here are the design goals in order to realize such a solution:

- an easily understandable and (friendly) usable approach for creating a coherent group of software artifacts for a domain (easy to encapsulate the know-how of a domain);
- flexible adaptation as a response to technological changes: a clear separation between the model and the technologies, but also a solid foundation to map on any software platform;
- a straightforward methodology to model and then to automate code generation for implementing and deploying a family of software products;
- simple ways for prototyping as an extension of standard tools (like Eclipse [35]) accepted on a large scale by the current research communities;
- an architecture designed for reuse integrating ideas from Domain Driven Development (DDD) [27], Aspect Oriented Programming (AOP) [66], Unified Modelling Language (UML) [80], Model Driven Architecture (MDA) [86], design patterns [38] [87], and generative programming [26].

## 1.3. Plan of the thesis

This chapter placed this work in the current research trends, defined the challenging problems that will be dealt with and our motivation, and then it introduced the proposition for a better approach.

Chapter 0 looks from the perspective of the paradigm shift that software engineering is experiencing today and introduces the state of the software modelling art. It shows the shortcomings of existing approaches and according to them it identifies and justifies features that a new approach should provide.

Chapter 3 presents SmartModels – an approach which relies on MDA and which is intended to contribute to strategies to increase software quality and productivity. Thanks to its meta-level, it clearly identifies the semantics of concepts used for the modelling of a given domain and, on the other hand, thanks to approaches of separation of concerns and of generative programming, it creates the basis for building the applications related to the target domain in a modular way. The presentation of the key aspects when modelling in SmartModels is accompanied by the methodology to describe the structure and behaviour of a model and by different casuistry which further explore and illustrate the flexibility and richness of the approach.

The meta-model of SmartModels is a core which acts like a foundation for various applications. Chapter 4 takes the discussion one step further and shows that this approach, which is based on customizable genericity, provides the needed expressivity for modelling product lines. Therefore, one of the most important applications, and also one of the main purposes of the approach, is to address the description of a line of products and this chapter gives full details on how to use SmartModels features in order to contribute for a better modelling of SPL.

For a better understanding of the interest of SmartModels, chapter 5 gives an overview of the implementation of a prototype, called SmartFactory, and of the platform and tools used to build it. SmartFactory represents the first practical validation of the approach.

Chapter 6 illustrates the approach and prototype through an example on how to use the complete solution. It presents a relevant case-study to evaluate the methodology to create business models and techniques to generate applications afterwards.

Finally, Chapter 7 concludes this work, comparing its original contribution to the state of the art and then wraps everything up by giving future perspectives to continue work.

659.988

# 2. The State of the Software Modelling Art

This extent of modelling engineering in the software domain has become an important concern for software developers, consultants and programmers. *Meta-modelling* may be defined as a method to design a stencil for producing something. Extending this to the specific case of software, we can say that the *meta-modelling of software* is a method of designing templates and patterns (in other words: *models*) for producing software.

In the past, the modelling process in software development, no matter how well implemented, has been very dependent on the skills of an individual developer. In [109], Tyrrell points out in the context of the importance of the software process that this approach can lead to three key problems.

Firstly, such a software project is very difficult to maintain. Imagine our software developer has suddenly resigned from the company and somebody else must take over the partially completed work. In a first scenario let us assume that there is some documentation describing the state of the work in progress. Maybe there is more than textual documentation, there is even a plan, with individual tasks mapped out and those that have been completed marked with different colours. In the other scenario, if the plan exists only in the developer's head and if there is no documentation then it is very probably the whole project may be compromised and even if the previous design was very good the new developer has probably to start from scratch.

Secondly, it is very difficult to evaluate correctly the quality of the finished product according to any independent assessment. If there are two developers, each working according to their own methods, and defining their own tests along the way, then there is no objective method for comparing their work either with each other or with a customer's quality criteria.

Thirdly, there is a huge overhead involved as each individual works out their own way of doing things in isolation. To avoid this, some way must be found of learning from the experiences of others who have already gone the same road.

Therefore, it is important both at the level of each organization, but more important, at the level of global standard, to define the modelling approach for a software project. At its most basic, this means to define an approach to model software, then to train the developers and offer support for the approach with efficient tools.

This chapter aims to present the state of the software modelling art:

- It briefly lays the cornerstone on software modelling as the first attempts to deal with increasing complexity, which are still adopted today in the industry, started with object-oriented programming;
- It summarizes the four classical modelling methodologies for application development;
- It introduces the Unified Modelling Language (UML) – Object Management Group's (OMG's) standard on modelling systems. It points out in more detail the elements which are extensively used in software modelling today, like: the diagrams, constraints and the extensibility of the notation ;

- It presents the Software Product Line (SPL) approach which is based on methodologies for strategic reuse of source code, requirement specifications, design models, etc., for building families of systems and modelling their artefacts;
- It addresses the focal concept of SPL: the software variability and mechanisms for its management.

## 2.1. Object-Oriented Programming (OOP)

The constant evolution of hardware and software systems is making it possible to develop information systems of steadily increasing complexity. The reason for the complexity of a system lies mainly in the number and variety of its constituent elements, the relationships and interactions between them and the differing, and often conflicting, needs of the user. The object approach provides some solutions to the problems of constructing complex systems and applications by picking out the stable elements so that they can be modelled by objects in a uniform manner [73], [13], [28]. This approach anticipates the need for an application to evolve with time and favours solutions that will ease this evolution and encourage the re-use of the software [93], [40].

Conceptually, an object is an abstraction from a real-world entity, for example an employee of a company (a rather simple object), or the site of a scientific laboratory or of a hotel (a rather complex object). It can have associated items of information (simple values or real objects), such as the last name and first name of the employee and the name and address of the laboratory at which he/she is located. Also, it can be manipulated by a set of operations specifically applicable to it, for example, the operation that assigns an employee to a site. Several principles underlie the process of mastering the complexity of a system. The first such principle is *abstraction*, which enables the behaviour of an object to be studied independently of its physical representation; the second is the *decomposition* of complex objects into simpler ones; and the third is *grouping* objects according to their mutual relations and interactions.

The object approach originated in object-oriented programming, of which the best-known languages are Smalltalk [61], C++ [98] [115], Java [4] and more recently C# [50]. It gained its first great success in the programming of graphical interfaces, now ubiquitous in computing. Since then it has been generalized for many fields, for example software engineering, distributed systems and databases, and then extended to handle system design and analysis. It brings important advantages, such as modelling of objects and modularity, re-usability and extendibility of code, which leads to higher productivity of the developers and better quality of the software produced.

The first true object-oriented language was Simula (1967) [37], which introduced the concepts of object and class from Algol; like Algol, Simula is a strongly typed compiled language. However, the first language to popularize the object approach was Smalltalk (1970). This can be regarded as combining the concepts of Lisp, from which it inherits its functional principles of an interpreted language, and of Simula, to which it adds the concept of *meta-class*. Smalltalk has proved able to satisfy the needs for flexibility raised by the development of graphical programming environments requiring rapid prototyping of cooperative user interfaces. It was used with success at Xerox PARC (the ancestor of the Macintosh) in the first graphical workstation.

With the arrival of workstations in the 1980s many object languages inspired by Simula or Smalltalk began to appear. Among compiled languages the best known are C++ [98], Objective C [52] and Eiffel [51], for all of which the language or the code generated is compatible with C. Most of the interpreted languages are extensions of Lisp, for example Loops and CLOS (Common Lisp Object System) [62]. It is interesting to observe that most of the popular existing languages are being extended in the direction of object orientation, for example Cobol and Ada – more precisely for the latter, Ada 94, which provides inheritance [39]. In the last decade the interpreted languages became of great interest, Java for example, because they give portability over several platforms due to the support of their virtual machine [37].

Having this in mind it is also important to see that there is a consensus of researchers on object-oriented methodologies for analysis and design of programs. They assert that although there is an important increase in complexity of today's requirements for software, we still do not take all the advantages of greater extendibility and easier maintenance that object technology can bring. Therefore, the object-oriented methods for modelling and design of the complex applications are now important challenges for many researchers.

## 2.2. Classical Modelling Methodologies

The more complex information systems become, the more the process of design and development must be rationalized. Some projects involve hundreds of people and can take several years to complete, and, although the costs of development are high, those of maintenance can be three or four times higher [22]. The financial stakes and risks are thus considerable. To reduce these costs and risks, computer scientists must use adequate tools and techniques. These will make the development of information systems an engineering activity at the same level as mechanical or civil engineering or industrial electronics. The design methods contribute mainly by providing a general approach to the problem or an action plan for solving the problem, models to synthesize solutions and quality-control tools.

Many design methods have been proposed since the 1970s ([53], [13], [55], [93]) and have contributed to the better definition of the steps of the design process, especially to the standardizing of the terms and representations used by designers and developers. The development of these methods has been influenced, inspired even, by the evolving technologies of programming languages, databases, computer networks and real-time systems. The new generation of object-oriented methods has itself been developed under the influence of object-oriented languages (OOLs) and databases.

The classical methods for the design of information systems suffer from many inherent shortcomings, in the models they support, the stages in the life cycles of the systems and the tools and techniques they use [75]. The data models were often designed to support classical business applications, handling only flat structures and providing few integrity constraints. The development cycle of these methods has seldom been carried so far as to cover all the phases of analysis and design, project management or programming tests, and so on. The software-engineering tools available on the market tend to be oriented more for documentation than for production: there is little added value between what designers put in and what they get as output, so that whatever productivity gain these tools provide is difficult to measure.

The four classical models for application development cycle are summarized below. All of them are entirely or partially adapted to the current object-oriented methods. Also, it is important to note that the order of their enumeration has a meaning: each one of the subsequent presented models tries to mitigate the shortcomings of the previous one [74]:

**The cascade model.** It consists of a succession of stages, going from the analysis of the requirements to the production of the final information system, with no real modelling on the way. The information system is not considered as a whole, but as a set of separate applications without global coherence between them. The analysis approach of each application consists in identifying the inputs and the outputs, and the transformations to be made from one to the other. In the design stage, a detailed technical specification is drawn up, in terms of files, algorithms and output forms. The implementation stage is the coding phase and the test stage is that of verification and validation.

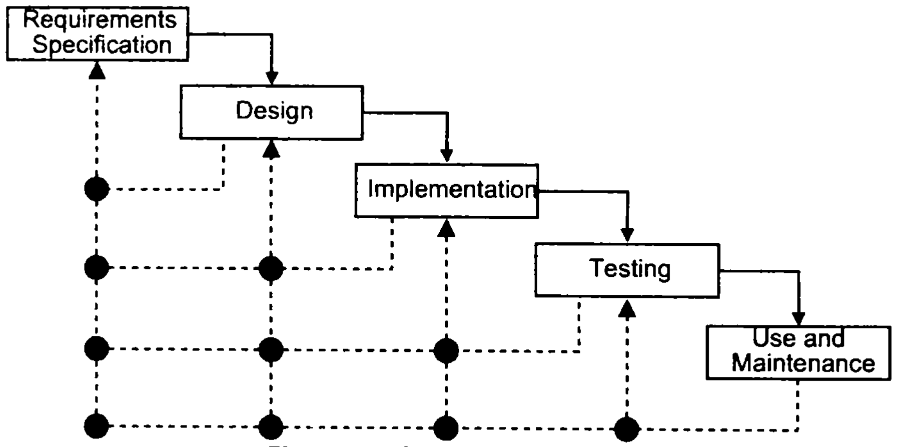


**Figure 1. The Cascade Model**

The main drawback of this model is its inability to deal with complex systems in which large numbers of applications are interacting with one another, not necessarily hierarchically. This model goes directly from the analysis of informally stated requirements to the highly technical phase of constructing a detailed specification. The test phase is of an overall nature and often aims only to establish that the functional operation of the system is correct, rather than that the original needs are satisfied.

Questions of several kinds can be raised concerning iterating the steps in the development cycle. Although there may be an argument for returning to a particular step so as to better adapt the technique to the requirements (or sometimes, the needs to the technique), it does not seem sensible to allow returns from any step to any other: it is not obvious that there is any sense in going back from the coding or testing stages to the requirements stage. And, although there may be value in some iterations, it seems essential to limit the amount of time that can be allocated for these, so as to avoid spending an excessive time in a particular stage.

Particular known methods that use this model are object-oriented design (OOD) [13], covering the second and third stages (see Figure 1), and object-oriented software engineering (OOSE) [55], covering the full set of stages.

**The V model.** This is a variant of the cascade model, introducing the concepts of system and component (subsystem) and employing a hierarchy of explicitly specified tests to give better control of the sequencing of the stages (see Figure 2).

The system component concepts improve the consistency of the design by making it hierarchical and modular, which in turn eases the development of complex systems. There is a final stage in which the complete system is validated with respect to the requirements. This illustrates very well the distinction between checking that the system has been constructed correctly, in the sense that it is logically consistent (*verification*), and demonstrating that it does what is required of it (*validation*). [97]

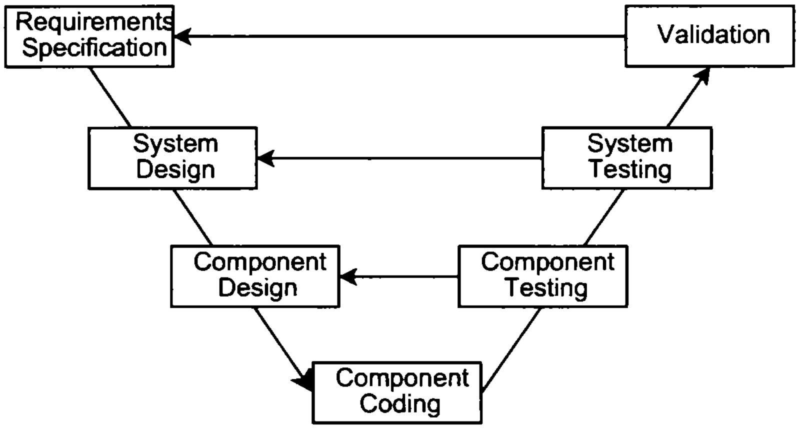


**Figure 2. The V Model**

Nevertheless, there are structural problems with this model, too. Although the logical verification by a bottom-up process from components to system is sound practice, the validation against requirements is left too late: it is expensive to find that a system is inadequate only after it has been built.

The GMT method [93] makes partial use of this model, using only the first branch of the V (descending phases).

**The spiral model.** The design proceeds by the analysis and prototyping of a series of projections of the future system, with the aim of validating them with respect to the functional needs.

This model [12] can be regarded as solving the shortcomings of the validation stage of the V model. It gives special importance to validation by performing this as early as possible in the cycle, and as often as necessary, by constructing a series of prototypes (see Figure 3).

The procedure is that, before the system that will be finally required is built, a number of relevant subsystem prototypes are built and each is validated with respect to the functional needs, the constraints imposed by the hardware and the software, and by the economic and strategic considerations that are relevant to the organization.

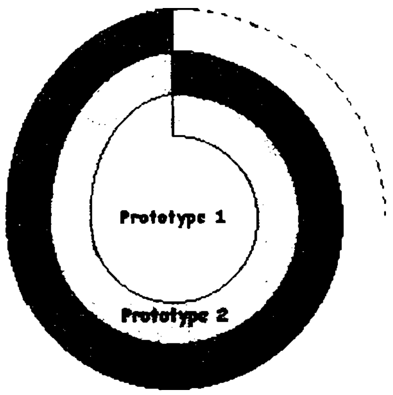


**Figure 3. The Spiral Model**

The design thus proceeds by the analysis and prototyping of a series of the future system projections, with the aim of checking that the designer has understood the real-world problem correctly and that the techniques used are sound. After this sequence of prototyping has been completed, and the appropriate conclusions drawn, the development of the final system proceeds along similar lines to the cascade model.

The main problem with this model is its use in practice, which will always seem expensive to the managers and the decision makers. Furthermore, there is no general agreement on either the size of the prototypes or the number that should be built, nor on what should be done with them subsequently - should they be simply discarded, or should they be used as kernels around which other functions are developed?

**The three-dimensional model.** This model was introduced by the Merise method, used very extensively in France [101]. It is the only model that takes explicit account of the database aspect, by clearly incorporating the ANSI/SPARC levels [1].

This model positions the development of the information system with reference to three axes (see Figure 4): the system life cycle, the project life cycle and the abstractions life cycle.

The information system life cycle concerns the lifetime of the system and the main periods after which major changes are made, such as increases in the workload (volumes of data or transactions handled), changes in technology (hardware or software) or structural changes (for example from centralized to distributed architecture). These changes determine actions to be taken during development or maintenance of the system. The abstraction cycle concerns the successive levels of the specification, going from the purely conceptual, independent of technology, to one that depends on one particular technical environment. The project life cycle - which could equally be called the decision cycle - is equivalent to the cascade model: it defines the sequence of phases through which the project is achieved. Thus, the design of the system is guided simultaneously and continuously by these three axes.

The originality of this model lies in the system life cycle and the abstraction cycle axes. The first enables the evolution of the system to be planned and any changes to be organized, whereas the second allows the conceptual solution to the problem to be formulated independently of the technical solution implemented and in so doing makes for improved portability of the system and greater scope for its evolution.

The problems raised by this model are the consequences on the one hand of the absence of any formalization of the system life cycle - more precisely, of any criteria by which this can be characterized - and on the other of the difficulty of giving any semantic interpretation of any of the planes defined by a pair of axes.

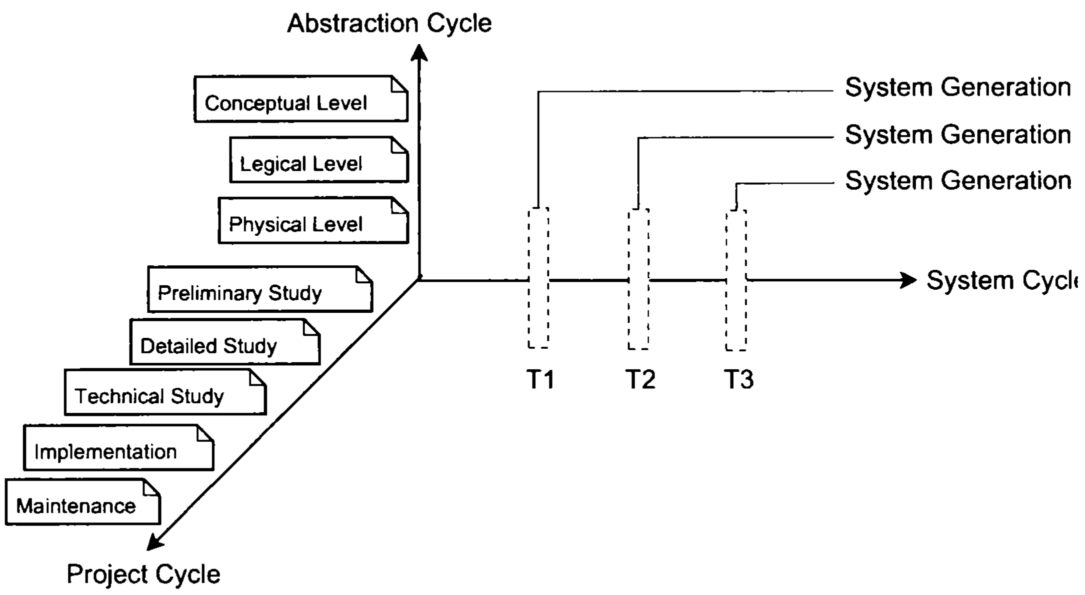


**Figure 4. The Three-Dimensional Model**

Many object-based methods take account of the abstraction cycle, for example OMT [93], OOSE [54], object-oriented analysis (OOA) and object-oriented methodology (OOM) [14]. However, none of them were related to Merise method.

## 2.3. OMG's Standard on Modelling Systems

Object methods, perhaps even more than languages and databases, have given birth to a proliferation of terminologies and graphical representations, which do not help the potential users to understand the methodologies or to compare them. Only standardization seems to overcome this difficulty and exactly at this point is OMG's most important contribution: *Model-Driven Architecture* (MDA) [86] and *Domain-Driven Development* (DDD) [27].

MDA proposes to separate platform-independent business-models (PIM) from the perspective of platform-specific design and implementation models (PSM). This is an approach that seems to keep its promise towards a better and more practical framework for software development. Therefore, OMG proposes a reference model [86] that can be extended and specialized; the model is abstract in the sense that it defines concepts but not their implementation. It defines a common semantics for the objects so as to specify their visible external features in a way that is independent of their implementation.

Speaking about the new OMG vision, Richard Soley and the OMG staff stated that: "OMG is in the ideal position to provide the model-based standards that are necessary to extend integration beyond the middleware approach. Now is the time to put this plan into effect. Now is the time for the *Model Driven Architecture*" [96]. This is the first step towards an extendible standard, bringing at least a common vocabulary for object systems.

Model technology is in the position to meet some of the challenges of the object-oriented methods, because it integrates aspect separation, homogeneous handling of functional and non-functional attributes, incorporation of different points of view (rules, services, processes, architecture, etc.) [96].

The next quote from MDA whitepaper is also challenging this direction: "It's difficult - in fact, next to impossible - for a large enterprise to standardize on a single middleware platform. Some enterprises found themselves with more than one because their different departments have different requirements, others because mergers or acquisitions created a mix. Even the lucky enterprise with a single middleware choice still has to use other technologies to interoperate with other enterprises and B2B markets. The middleware environments that are most visible today are CORBA, Enterprise JavaBeans, message oriented middleware, XML/SOAP, COM+ and .NET. However, over the past decade or so, the middleware landscape has continually shifted. For years we've assumed that a clear winner will emerge and stabilize this state of flux, but it's time to admit what we've all suspected: *The sequence has no end!* And, in spite of the advantages (sometimes real, sometimes imagined) of the latest middleware platform, migration is expensive and disruptive. We know an industry standards group that, having migrated their standard infrastructure twice already, is now moving from their latest platform to XML." [86]

Technology neutral models of systems can be mapped to implementations that use a variety of middleware technologies. It seems that it is time for *models everywhere*, but the road to model engineering takes time. MDA is not a new technology, but a way to deal with new emerging technologies. MDA is about integration and evolution management. Decoupling the business part from the technical part of information systems is a long-term trend.

### 2.3.1. Unified Modelling Language

Now, **UML (Unified Modelling Language)** is the standard of the OMG (Object Group Management) for modelling software systems. It proposes a sum of notations, in the form of diagrams, for the systems design and the documentation. The UML diagrams model the software according to different standpoints: functional view, static view, dynamic view and implementation and operation view. UML is primarily intended to model only one software product at a time. However, as this chapter develops, we will see that several research domains were interested in the use of UML for modelling of the SPLs (Software Product Lines - see section 2.4). This is explained by two major reasons:

- UML is a largely adopted standard in the industry and now there is much tool support for this approach;
- UML defines standard mechanisms of extensions making it possible to extend and adapt its notations and semantics to a particular domain. The *stereotypes* and the *tagged values* are examples of these mechanisms.

The main objective of this section is to present the state of the art based on books and research papers around the handling of the product lines in UML. It begins with a short reminder of UML and then it introduces SPL concepts.

UML notation [80] [81] [83] constitutes an important stage in the area of notations used for analysis and object oriented domain modelling since it represents a synthesis of the most used methods: OMT (Object Modelling Technique) [93], OOD (Object-Oriented Design) [13] and OOSE (Object-Oriented Software Engineering) [54]. Since the first version of UML, this standard proposed under the OMG (Object Management Group) umbrella has not ceased developing. However, the major release of UML is the UML 2.0 standard, [80] [81], adopted in August 2003. UML 2.0 represents an important step in sustaining the growth of the current software complexity, on the one hand through the support of the new vision of the OMG of acknowledging MDA (Model Driven Architecture) [96], and on the other hand through the support of new technologies, especially the software components approach.

In this version of UML, semantics are based on meta-modelling. Since its first versions, UML standard was characterized by a meta-modelling approach defined through its semantics. In this context it is important to assert that *a meta-model is a precise definition of the constructs and rules needed for creating semantic models* [118]. Therefore, the UML meta-models define the strict structure which any UML model has to take on. The meta-model specification of UML1.x is defined in only one document, while the UML2.0 standard is now divided into two documents: UML2.0 Infrastructure [81] and UML2.0 Superstructure [83]. UML Infrastructure describes the fundamental entities used for the definition of its infrastructure library (*InfrastructureLibrary*). Its scope meets the following requirements:

- formal definition of a common object analysis and design (OA&D) meta-model to represent the semantics of OA&D models, which includes static models, behavioural models, usage models, and architectural models;
- IDL specifications for mechanisms for model interchange between OA&D tools. This document includes a set of IDL interfaces that support dynamic construction and traversal of a user model;
- a human-readable notation for representing OA&D models. This document defines the UML notation.

UML Superstructure re-uses and refines the infrastructure library and defines the meta-model itself, as seen by the users to bring agreement on semantics and notation:

- a formal definition of a common MOF (Meta Object Facility) based meta-model that specifies the abstract syntax of the UML. The abstract syntax defines the set of UML modelling concepts, their attributes and their relationships, as well as the rules for combining these concepts to construct partial or complete UML models;
- a detailed explanation of the semantics of each UML modelling concept. The semantics define, in a technology-independent manner, how the UML concepts are to be realized by computers;
- a specification of the human-readable notation elements for representing the individual UML modelling concepts as well as rules for combining them into a variety of different diagram types corresponding to different aspects of modelled systems;
- a detailed definition of ways in which UML tools can be made compliant with this specification. This is supported (as a separate specification) with an XML-based

specification of corresponding model interchange formats (XMI) that must be realized by compliant tools.

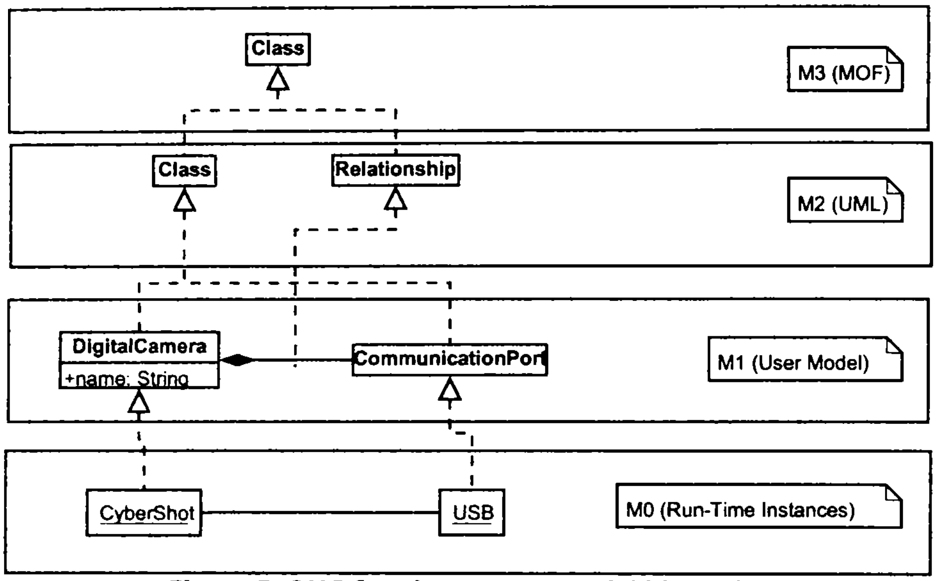


**Figure 5. OMG four layer meta-model hierarchy**

The approach of meta-modelling adopted by the OMG is known as a hierarchy on four levels [81]:

**Meta-meta-model level (M3).** M3 is the meta-meta-model level; it defines the meta-model language specification. The MOF [84] is an example of a meta-meta-model.

**Meta-model level (M2).** M2 is the meta-model level. The UML meta-model is at this level and it is defined using MOF, which means that the concepts of the UML meta-model are instances of MOF concepts. Figure 5 shows two meta-classes of the UML meta-model: *Class* and *Relationship*.

**Model level (M1).** M1 corresponds to the UML user models level. The concepts of a UML model are instances of the UML meta-model concepts. Figure 5 shows an example of a class diagram of an application for a digital camera factory containing two classes: *DigitalCamera* and *CommunicationPort* linked by an UML association. The two classes are instances of the *Class* meta-class and the link is an instance of the *Relationship* meta-class of the UML meta-model.

**Objects Level (M0).** M0 corresponds to the run-time level of objects. This presents two objects: *CyberShot* and *USB*, instances of the two model-level classes: *DigitalCamera* and the corresponding *CommunicationPort*.

The UML meta-model [83] is described using a part of the notation of the UML itself. It uses the following concepts:

- UML Classes – to describe the meta-classes;
- Attributes – to describe the properties attached to a meta-class;
- Associations – to describe bonds between the meta-classes;
- Packages – to group the meta-classes by domain.

### 2.3.2. The OCL constraints

The UML meta-model specifies the structure which any UML model has to adopt. In other words, it specifies structural constraints on these models. UML includes OCL (Object Constraints Language) [82] as an additional option to strengthen the structural constraints of the UML models by adding invariants on the classes of the UML meta-model. Therefore, OCL constraints on the meta-model level represent rules of conformance on the UML models. They are expressed at the meta-model level and they are evaluated on all the entities of the UML models, actually instances of the entities of the UML meta-model.

The OCL constraints are also used to express properties of an UML model (M1 level). They are used to describe invariants, pre-conditions and post-conditions for methods and guards for state machine transitions (Abstract State Machines). The constraints expressed on the M1 level are evaluated and checked on the models of objects (M0 level).

### 2.3.3. The diagrams

The UML notation is described in the form of a set of diagrams. The first generation of UML (UML1.x) defines nine diagrams for software documentation and specification. UML2.0 Superstructure [83] added four new diagrams: composite structure, package, interaction overview and timing diagrams. They are organized in two main categories:

- **Structure diagrams:** organizes the hierarchy of class diagrams, component diagrams, composite structure diagrams, deployment diagrams, object diagrams, and the package diagrams.
- **Behaviour diagrams:** activity diagrams, state machines diagrams, use case diagrams, communication diagrams (the new name of the UML1.x collaboration diagrams), interaction diagrams, sequence diagrams, timing diagrams, and protocol state machines diagrams.

This thesis heavily uses the UML class diagrams and this is why next paragraphs will briefly introduce them together with the use-case diagrams. The fact that there are several works in the current research communities around modelling product lines with use-case diagrams [111], motivated the choice to also briefly introduce them, too. All the UML diagrams are documented in [83].

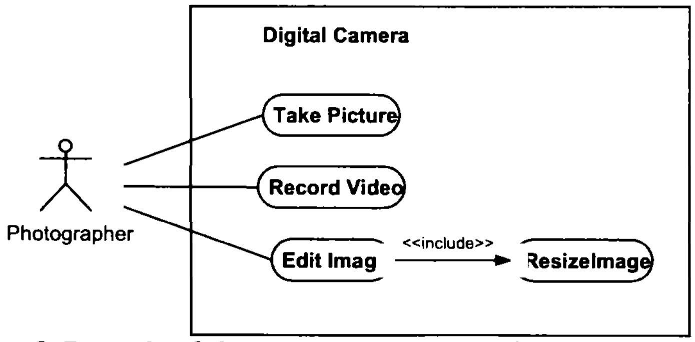


**Figure 6. Example of the use cases and actor for a digital camera**

The **use case diagrams** represent the principal contribution to UML of Ivar Jaccobson and his method OOSE [55]. The scope of the use cases is to capture the requirements in terms of services which a system must ensure. The use case diagrams define two main concepts: *actors*, and *use cases*. An actor is an external entity of the system which can initiate one of these use cases. A use case is a functionality provided by the system.

Figure 6 shows an example of use cases and actor for a digital camera. The main actor, *Photographer*, can initiate three use cases of the system *TakePicture, Record Video* and *EditImage*. The use case diagrams make it possible to define two types of relationships between the use cases:

- *extend*: a relationship from an extending use case to an extended use case that specifies how and when the behaviour defined in the extending use case can be inserted into the behaviour defined in the extended use case.
- *include*: an include relationship defines that a use case contains the behaviour defined in another use case – this means that the service specified by the second is included in the service of the first (*EditImage* may include the *ResizeImage* use case).

A use case can extend a case of its parent use – this means that the service of the first is a specialization of the service of the second. The relationships between the use cases are noted like arrows of dependences with the key words `<<include>>` for inclusion and `<<extend>>` for extension.

The **class diagrams** define the classifiers structure of a system. Besides its name, a *Class* is described by its attributes and methods. The class diagram in Figure 7 shows two classes: *DigitalCamera* and *Memory*. The *Memory* class has two attributes: *capacity* and *speed*. The *DigitalCamera* class has two attributes: *name* and an enumeration of *communicationPorts,* and a couple of methods. Besides the structure of the classes, the class diagram allows to represent also the relationships between these classes (for example, UML associations). An association may contain additional information: the name of the relationship, the role of each class in the association and cardinalities.

Figure 7 illustrates the composition relationship between the two classes. An instance of the *DigitalCamera* class can contain a set of *Memory* class instances (cardinality *). However, an instance of the *Memory* class may reference to one and only one instance of the *DigitalCamera* class (cardinality 1).

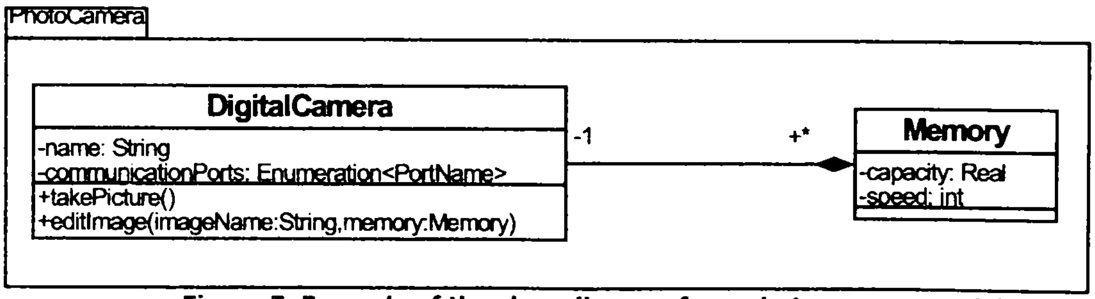


**Figure 7. Example of the class diagram for a photo camera model**

UML also provides mechanisms for abstraction in the class diagrams through the notions of inheritance, abstract classes and interfaces. *The Inheritance* (specialization/ generalization) is a mechanism which enables a sub-class to inherit the attributes and methods of another class, called super-class. A sub-class can redefine the implementation of one or several methods of the super-class. An *abstract class* can contain methods whose bodies are defined by its heirs (sub-classes). The name of an abstract class in a class diagram is marked in italic. An interface declares only the signatures of the methods without any implementation (without a body). The notation of an interface includes the `<<Interface>>` stereotype.

Packages is a mechanism for organizing classes. A UML package is a namespace. Figure 7 presents an UML package called *PhotoCamera* which contains the two classes: *DigitalCamera* and *Memory*.

### 2.3.4. Extensibility

At the beginning, the UML notation was intended for the modelling of any type of systems. However, certain systems in specific domains are characterized by properties which require the addition of auxiliary information. For example, the real time systems domain needs also to specify services quality properties. The UML standard notation does not enable to add such information. This need determined the designers to create extensions of this notation. Besides the specific domains, there is also the need to adapt UML to a particular technical platform. For example, in order to model a system which will be implemented on a EJB (Enterprise Java Beans) platform, the meta-programmer needs to say that a particular class plays the part of an EJB component (Bean).

To meet these particular needs, the OMG has introduced new mechanisms, called *extension mechanisms,* which enabled the UML to specialize and adapt for different purposes to various specific domains, platforms or methods. The extension mechanisms introduced since version UML 1.3 are the *Tagged Values*, the *Stereotypes*, and the *Constraints*. The *Profile* was introduced as a concept which groups the three preceding mechanisms. The OMG also created standard profiles for some particular domains. For example: UML profile for the real time systems [79], UML profile for the company distributed applications (EDOC) [78] and also UML profiles for the EJB and CORBA platforms.

In addition to these standard profiles, the users can define their own profiles. Certain works, such as the Objecteering Softeam approach, have enabled their implementation and integration into the modelling environment (in order to express and manage requirements, build complete and accurate UML models, produce reports and documentation and automate application code production for Java, C++, C#, SQL, CORBA and Fortran) [95]. The profiles in the UML 2.0 standard are part of the infrastructure document [81]. The *Extension Mechanisms* chapter in UML 1.x [80] is replaced by a chapter called *Profiles* in UML 2.0 Infrastructure document.

Next paragraphs will briefly describe two important concepts of this document: the stereotypes and the profiles. They are described in mode detail the UML Infrastructure document [81].

The **stereotype** is the basic mechanism for extension in UML. It defines how a particular existing meta-class of the UML meta-model can be extended to enable the use of a terminology or a notation specific to a particular domain or platform in place of, or in addition to, the ones already used for the extended meta-class. A stereotype is a kind of a particular meta-class of the meta-model and extends it through *extension* [81].

The tagged values introduced in UML1.x are considered in UML2.0 as properties of the stereotypes. When a stereotype is applied to an entity of the model (an instance of the meta-class on which the stereotype is defined), this entity will be noted with `<<label-stereotype>>` and the values of its properties, if they exist, will be noted down as tagged values associated with this entity. A tagged value has a *name* and a *type*.

Figure 8 shows an example of the definition of two stereotypes [81]: *Camera* and *Turist*. The *Camera* stereotype extends two meta-classes: *Class* and *Component*, and it defines a *hasAutoFocus* property as a tagged value whose type is *boolean*. The notation of extension is an arrow pointing from a Stereotype to the extended Class (see Figure 8).

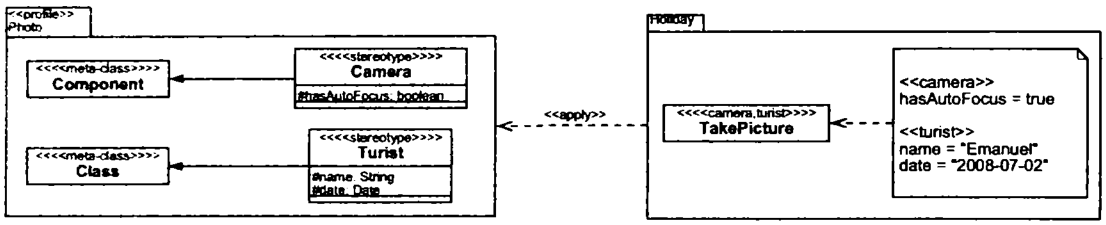


**Figure 8. UML 2.0 profile example**

By definition, a **profile** extends an *existing meta-model* or another profile [81]. The existing meta-model can be the meta-model of the UML or another meta-model based on the infrastructure document. The name *UML profile* is used to indicate a profile whose reference model is a UML meta-model. The UML profiles can be considerred as *dialects* of the UML language [81]. An UML profile contains a set of stereotypes and tagged values.

Figure 8 shows an example of the UML profile called *Photo* which contains the two stereotypes *Camera* and *Turist*. The UML 2.0 profiles are using the same notation as for UML packages but adding the `<<profile>>` stereotype. The *Holiday* package is an example of an user model based on the *Photo* profile. The *TakePicture* class is defined with the two profile stereotypes. The associated tagged values of the stereotypes are represented as UML notes (see Figure 8).

In addition to the stereotypes and tagged values, an UML profile can also contain *constraints* and *rules*.

The constraints define and control the semantics of the meta-model entities of the reference profile. They can be attached to the meta-classes or to the stereotypes from the profile.

The rules in an UML profile describe its usage. For example, they can describe how the code is generated starting from models based on this profile or how the models are transformed from other models, etc. The rules in a profile are often defined by using a language of model transformation. For example, the Objecteering Software team proposes in [95] a language called J for the definition of a profile's rules.

## 2.4. Lines of Software Systems

The "*Software Engineering*" term was popularized at the end of '60s to answer the growth of software complexity. Its main goal was to define and propose more flexible and efficient methodologies for the development of software solutions. Its domain of research evolved towards approaching the new problems which deal with the penetration of software into the wide diversity of industries which require software applications.

Therefore, now it is no longer important to develop only one software solution applied to a particular problem, but rather to design and develop a line (or a family) of software which takes into account features that may vary and makes it possible to minimize the costs and time of realization. The features which may vary can be technical (use of a variety of resources associated with the software), commercial (creation of several versions, starting from a limited version to a complete one), or cultural (software intended for several countries). For example, one may have to adapt an integrated software into the mobile phones which must support several standards of communication and a variety of languages (for example, [70] explains that the Nokia phones must support more than 60 languages).

The concept of Software Product Lines (SPL) is not completely new, as David Parnas in [88] already started to study the families of programs since 1976. However, this paradigm did not become popular among the software research communities until the last decade. Then, a community began to gather through different European projects such as ESAPS [91], CAFE [90] and FAMILIES [92]. In the United States, the Software Engineering Institute (SEI) created a special department which is involved in product lines engineering [77]. Also, every year there are conferences, like SPLC (Software Product Line Conference) and PFE (Product Family Engineering) which are entirely devoted to this approach (the tenth edition of SPLC which was organized in 2006 merged these two former events - http://www.splc.net/).

The good news is that in the literature there is consensus on the definition of the software product line approach:

*A software product line is a group of products that share a common, managed set of features. The products satisfy the specific needs of a particular market or mission, and are developed from a common set of core assets in a prescribed way.* [31] [76].

*A domain is a business field or a technology or the know-how described by a set of concepts and terms understandable by the users of that domain.* [76].

In other words, Software Product Line Engineering is the discipline of engineering a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. The organizational characteristics required to successfully carry out software product line engineering will vary according to an organization's needs. There is a minimum list of abilities which are most needed in using software product lines: architecture, programming techniques and development tools [29].

The Software Product Lines (SPL) approach – also called Product Family Engineering is a transposition of the technological lines of production into the world of the software. Frank van der Linden justifies the origin of two different names due to geographical influences [110]: *product line* definition is used in the United States of America whereas *product family* is used in Europe. This thesis will use both of them as synonyms.

The SPL goal is to minimize the costs of software implementation for a particular domain and the idea is simply this: instead of developing each application of the given domain separately, this approach proposes to construct all the related applications from reusable components.

The heart of the product line approach is based on a methodology for strategic reuse of source code, requirement specifications, software architectures, design models, systematic product line creation and improvement, components, test cases, and the processes for building families of systems and modelling their artifacts.

All these issues will represent the evaluation criteria this thesis will use in order to get ideas and evaluate the approach that it develops for the modelling in software engineering [72]. The main contribution of this thesis is to document and develop an approach to help the meta-programmers automatically generate flexible and efficient templates for lines of software products.

The first problem related to the SPL approach is the need for a design of the domain architecture in order to make it possible to define several products. The members of a product family are characterized by their common elements (also called commonalities) as well as by their differences (also called variabilities). The management of variability is one of the key aspects of the product lines modelling and implementation. In a technological line for vehicle production, cars are manufactured starting from a set of common elements (wheels, board, windows, etc), but can include some properties which differentiate them (the number of horse-power engine, presence or not of air-conditioning, etc).

In the world of software, the differences can come out in a similar way, according to technical choices (use of a particular tool or technology), based on the commercial (creation of a limited version), or regional (products intended for specific countries) choices. Another important issue when using the product line approach to build a software product (addressing also the product derivation) is to set some properties with respect to the variability supported by the product line. Obviously, certain option combinations are incompatible.

Here is an analogy which exemplifies the previous mentioned principle: a car is normally equipped only with one engine, and it is then necessary to choose between a petrol and diesel engine. In the same way, a particular choice at the time of the software product derivation can exclude other options. For example the choice for a two door car will normally exclude the possibility of choosing to have rear sliding windows. Therefore, a product line must integrate constraints to keep its coherence and to facilitate the choices at the time of derivation.

The above definition of the product line describes the products, members of the product line, by a set of common properties (commonalities), and also by their differences (variabilities).

*Variability is the ability of a system, an asset, or a development environment to support the production of a set of artifacts that differ from each other in a pre-planned fashion.* [8]

*Commonality represents the set of property values which apply and describe all the products and members of the product line.* [117]

Therefore, the concept of variability is used to group the properties which differentiate the products of the same family. The number of supported languages on the mobile phones domain can be an example of variability in SPL. Existing defined methods and tools supporting product line variability management typically emphasize either the feature or the architecture level. There have been attempts to combine these aspects, but no widely accepted method is available so far.

The management of the product variability in a family is the primarily activity when developing product lines. The second activity relates to the construction of a particular product (often known as derivation of a product) which consists in choosing specific values for the properties which are part of the variability defined in SPL. One characteristic of SPL is that certain choices are incompatible between them while others are dependent. A specific choice at the time of the derivation of a product can exclude or require other choices. A product line must thus also manage constraints making it possible to facilitate the choices when we derivate products.

The software engineering research communities have been more and more interested in the last decade to find solutions for the problems of SPL. Several important projects developed both at the level of the model and on the level of the code by using various technologies like object-oriented design [3] [57] [63] [99] and generative programming [2] [26] [9].

Even if this work is based on latest technologies and techniques which witnessed a remarkable success on the domain of the software engineering, the control of the code becomes an impossible mission with the exponential growth of the software complexity (today it is not surprising to find systems with more than a million lines of code).

Linda Northrop points out a couple of key obstacles and drawbacks related to the adoption of software product line approach [77]:

- cost of entry into software product line is non-trivial, both in terms of money and time;
- inadequate scope definition: large scope means too much feature variation between products and their family of products, leading to bad economies of scales (in reuse);
- to achieve successful software product line adoption, you need to consider both the product and process propositions.

The domain of system-family engineering is a long run challenge which can only be faced by systematic software engineering approaches, structuring systems into families and applying code generation for the efficient assembly of the product line members.

Independent from the code, modelling makes it possible to better control this complexity. It is no longer a question of handling the code of the system, but rather to handle a group of related models describing the system in an abstract way. Modelling is the base of several important methods of analysis and design such as OMT [93], OOD [13] and OOSE [55] which gave birth to UML (Unified Modelling Language) [80] [83] [81], "the current industrial standard on modelling". From then on, it was born a new field of software engineering based on modelling (model engineering), Model Driven Engineering (MDE) [86]. This thesis is in the context of the modelling of product lines and proposes new approaches for their handling.

Today there are many projects focused on handling of SPL in UML [47] [112] [58] [41] [42] [5] [6] [30] [33] [34] [21] [20]. In order to allow modelling of SPL and to take into account the new challenges on how to model variability, how to derivate products and how to handle the management of constraints, the majority of projects launched them into the use of standard mechanisms to make UML extensions. However, I believe that by studying more deeply these approaches, there are still many issues which are not covered and which shows that SPL handling in UML is not mature enough:

- the first reason is that the majority of existing efforts use only two aspects of UML to model SPL: use-cases [112] [47] [42] [58] and static models [116] [30] [21] [20]. In this context I would mention that few works refer to the dynamic aspect of SPL, although, besides use-cases and static diagrams, UML includes sequence diagrams and state diagrams and other tools which make it possible to model also the dynamic behaviour of the systems. But as we will see it later in section 3.1, the existing tools do not offer powerful mechanisms to express variability;
- the second reason takes into discussion the need for support of the product derivation which is an important aspect in an approach SPL oriented. Some projects are only dealing with model variability and do not refer to the product derivation [30] [41]. This means that they limit the utility of their work just to a descriptive goal and the objectives of the works are limited to a better way to document the software architectures using UML models, but there is still a gap between models and source-code. The effort of [5] [6] [33] refers to the derivation of products, but do not formalize it and do not propose a solution for its implementation;
- the third reason draws attention to the management of the constraints in SPL. As it was pointed out above, SPLs are characterized by a set of constraints which guide the derivation of products. There are only two approaches using UML ([21] and [34]) which attempt to implement constraints on SPL. However, none of these use OCL (Object Constraint Language) [46] to describe the constraints of SPL although OCL is part of UML OMG standard.

### 2.4.1. ESI's Work on System Family Engineering

ESI (European Software Institute) [110] has a couple of important initiatives on promoting the developing of software product line approach in building family of applications. In this framework, a large community of researchers have noticed today's paradigm in software engineering: from the engineering of single systems to the engineering of multiple systems or system-families: the analogy in the automobile industry is going from a single product for every customer, like the Model T Ford, to a product-line production strategy to satisfy diverging customer needs [91].

ESAPS (Engineering Software Architectures, Processes and Platforms for System-Families) [91] project motivation presents system-families as strategic business assets. The structuring of systems into system-families allows sharing of development effort within the system-family and as such counters the impact of ever growing system complexity. This makes it possible to sustain the rate of product innovation, while keeping guaranteed levels of overall system performance and quality. The fundamental concept of a system-family is a domain specific product architecture based upon a layered set of platforms. This is supported by a software engineering process focused on pervasive reuse.

A system-family is defined as a group of systems sharing a common, managed set of features that satisfy core needs of a scoped domain. The idea behind a system-family approach is to build a new system or application from a common set of assets (domain model, reference architecture, components) defined from earlier developed systems belonging to the same line. A software asset is a description of a partial solution. It might be a component, known requirements or design elements that an engineer uses to build or modify a software product.

The partners participating in ESAPS have researched software technologies for system-families for several years in two European projects: ARES and PRAISE [91]. Experiences gained so far are very significant. The system-family methodology, resulting from the PRAISE project identifies two engineering types:

- the *domain engineering* in charge of the analysis, design and the management of the domain assets (comprises the Application Family Engineering, Reference Architecture, and Component System Engineering);
- the *application engineering* in charge of the development of a new product using the domain assets (known as Application System Engineering).

Defining the assets from existing systems is not an easy task and requires the integration and refinement of extensive amounts of domain knowledge.

A step forward is the next project called CAFÉ (From Concepts to Application in System-Family Engineering) [90]. Its main drive is to go from ESAPS [91] concepts to application, so that these ideas can be applied in concrete projects by developing methods and procedures from these concepts. This approach underlines the concerns originating from several sources which influence software development. The sources are grouped into four categories:

- Business concerns - the way that profit is made by the resulting products,
- Architecture concerns - the technology needed to build the system,
- Organization concerns - the organization in which the software is developed,
- Process concerns - the responsibilities and dependencies during software development.

Changes in one type of concern will change the way the other concerns are dealt with. A well-designed software development has well-designed management for these concerns.

FAMILIES (is the acronym for FAct-based Maturity through Institutionalisation. Lessons-learned and Involved Exploration of System-family engineering) [92] is the final project in a row, following ESAPS [91] and CAFÉ [90]. It concentrates upon maturity, institutionalization, business relevance, standardization and dissemination. Its work concentrates on building:

- a framework for reuse which deals with questions like when, why and how a family approach has to be introduced, adding an integrated approach to combine existing legacy assets into a family, or even to a system population;
- patterns, styles and rules related to satisfaction of business related quality requirements in the family, accompanied by quality models, supporting processes, check lists, questionnaires and approaches towards standardization of quality of service requirements;
- a methodology in the context of MDA [86] standardization frame (process, tools, guidelines, and examples) supporting the separation of the domain aspects, the technical aspects (quality of services) and the technological aspects (platforms) in consistent models.

### 2.4.2. An approach adopted in the industry: Domain and Application Engineering

The product line approach was adopted from its very beginning in the industry stimulated by the diversity of the software variation factors in the business domains. The SEI (Software Engineering Institute) already published several industrial experiments proving its success [67].

[70] [71] shows how Nokia chose the product line approach to manage the diversity of the mobile phones software. [71] shows that Nokia must launch between 30 and 40 new products per year in order to keep its share of the market. If the production of every phone would start from scratch then it would be very difficult and costly to keep up with the market.

Further on, Nokia must also answer several other factors of variation between its products such as the user language interface. Indeed, the Nokia products support 60 languages [71] and each language has its own characteristics: most of European languages are based on the Latin characters and are displayed from left to the right; the Arabic languages must be displayed from right to the left and the characters need to be linked so they form different special signs; and then the Chinese languages which also have their own characteristics. The Nokia products must also be compatible with different standard of digital transmission such as GSM 900, GSM 1900, TDMA, etc. [70].

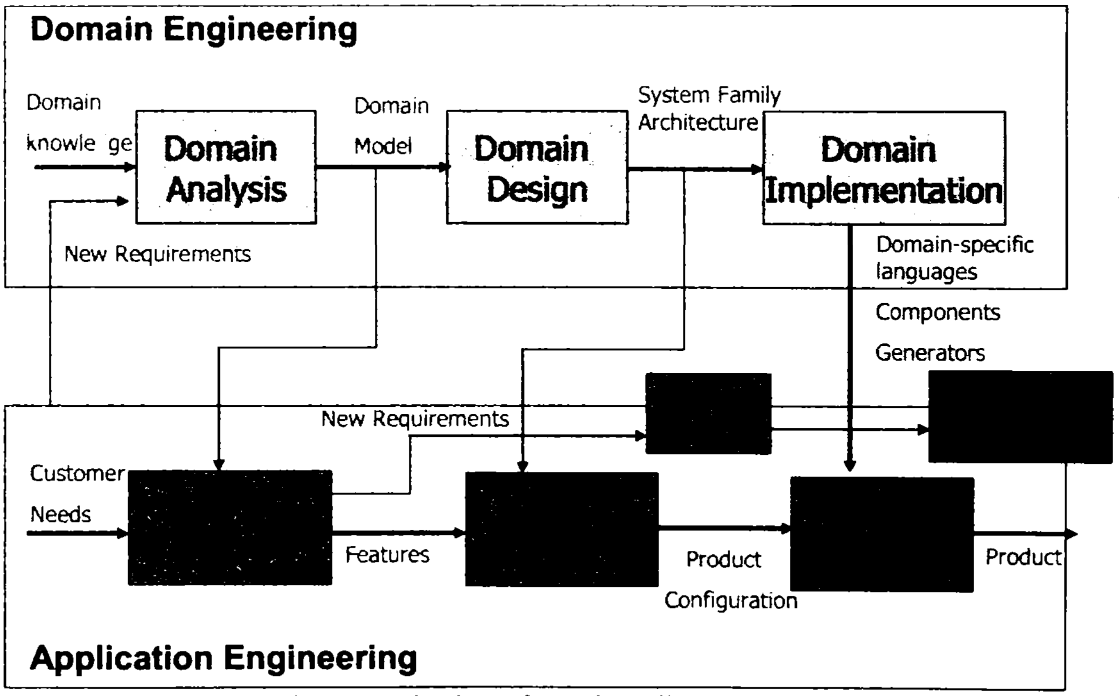


**Figure 9. The Domain and Application Engineering**

According to the definitions in the research communities [26] [91] [90], the software product lines engineering distinguishes two levels (shown in Figure 9 - [91]): Domain and Application Engineering.

**The Domain Engineering** (which includes the Application Family Engineering, Reference Architecture, and Component System Engineering) is in charge of the analysis, design and the management of the domain assets (an asset is an element which makes it possible to develop an application, for example a requirements document, a model, code, etc.) which will be reused for building products; it develops tools which are used for creating products. At this first level, three activities can be distinguished [26] [91] [90]: analysis, design, and implementation of the domain.

The goal of the *domain analysis* phase is to study the product line domain and to identify the commonalities and variabilities between the products. There are several methods for the domain analysis, among which one of the best known is FODA (Featured-Oriented Domain Analysis) [60]. FODA is a method based upon identifying the prominent or distinctive features (or properties) of a class of systems, specified in the form of a tree whose nodes represent the domain features and the arcs describe ways of composition between them. FODA distinguishes three categories of properties: alternative, optional, or mandatory. Mandatory features represent baseline features (they are obligatory properties for every product line instance) and their relationships. The alternative (define the scope for an exclusive-or choice of features) and optional (may be present, or not, in a product line instance) features (which apply only to some products) represent the specialization of more general features (i.e., they represent what changes are likely to occur in different circumstances).

Figure 10 shows a feature diagram example for a car product line [26]. Each property in the diagram corresponds to a concept of the domain. The mandatory features are represented by lines ended with filled circles, while the optional features are represented by lines with empty circles. The *Air-Conditioning* feature in Figure 10 FODA model is optional. There are two types of transmission in SPL of cars: *Manual or Automatic* and these are specified by the alternative feature called *Transmission*. An alternative feature is represented by an angle-arc which unites the lines of all alternative features.

The goal of the *domain design* is to establish generic software architecture for product lines. There is no consensus on the definition of a domain architecture and consequently for the product line architecture. Clements & al. [67], defines the product line architecture as a standard architecture which contains a set of components, connectors and constraints (it provides a set of techniques, not a prescriptive method for architectural design).

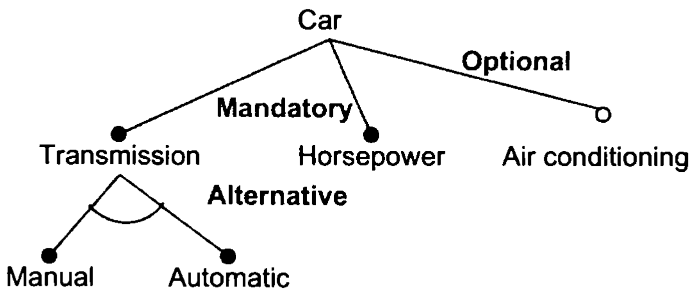


**Figure 10. FODA Feature Diagram Example**

Applying this to product lines, the architecture should be a reference from which the architecture of each product is derived. The variability identified during the domain analysis must be specified explicitly in the architecture of the product line.

The *domain implementation* consists in implementing the generic architecture defined in the domain design as components which will be reused in the application engineering in order to build particular products.

**The Application Engineering**. The Application Engineering (also known as Application System Engineering) is in charge of the development of a new product using the domain assets. It uses the results of the Domain Engineering for building, (also called derivate) of a particular product. It reuses tools previously developed to build particular products (see Figure 9). As mentioned above, the results of the domain engineering (feature models, generic architecture, and components) contain variability - therefore, the derivation of a particular product involves decisions (or choices) associated with these variation points. The concept of *decision model* [5] is used to capture and record the decisions necessary (adds a set of instructions) to the derivation of products.

## 2.5. Software Variability

The commonality and variability are the focal concepts in the software product lines. This section will mainly detail the concept of variability, but it also addresses the concept of commonality. This is justified by the fact that the management of variability requires more effort than that of the commonality. In reality, the common properties in SPL were identified and used like any other properties for the construction of the products until now. However, variability requires not only special attention for the process of its identification and description, but also specialized mechanisms for its management. Even if the product line approach is a new paradigm, the management of software variability is not a new problem.

There are several techniques used today for the design and the programming phases which make it possible to manage the variability [3] [99] (see also section 2.5.3). Nevertheless, apart from the context of product lines, variability also relates to one product, it is part of the product and it is solved after the product is delivered and installed in its execution environment.

In the context of the product lines, variability must be explicitly specified and it is part of the product line. Differing from the variability of only one product, variability in the product lines is solved before the product is delivered or installed in its execution environment. In [5], Atkinson & al. call the variability contained in only one product the run-time variability and that contained in the product line as development-time variability. This thesis addresses the variability of SPL, therefore it is the variability dealt with before the delivery of the product installation.

### 2.5.1. Dimensions of Software Variability

Variability can be viewed as consisting of two dimensions: [15] [59]: time and space (see Figure 11). The *time* dimension relates to the variation in time of one particular software product. Figure 11 shows the products evolution in time from a version to another. The *space* dimension relates to the variation (differences) between several products of the same family. Some software elements can be used in several products and their variation relates mainly to variations of functionalities; therefore the products may vary in the functionality that they support. This thesis considers only the space dimension, thus it studies the functionality variation related to the multiple contexts the products will be used.

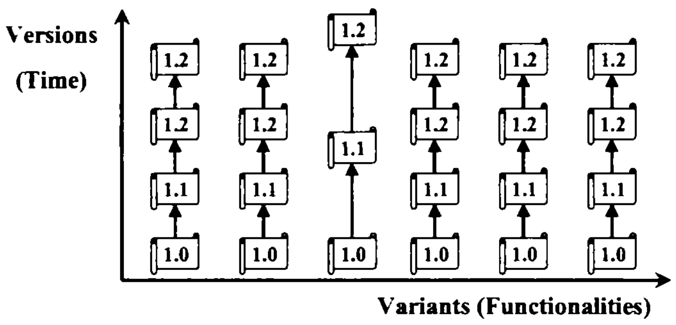


**Figure 11. Space and Time Dimensions in Variability**

### 2.5.2. The Variation Point

In the product lines approach the variability is identified during domain engineering and it is introduced by what is called variation points. Jacobson & al. [54] define a variation point as follows:

*A variation point identifies one or more locations at which the variation will occur.* [54]

A variation point can be seen like a decision point [47] with a couple of variables called possible choices. The *Transmission* node in the FODA diagram of Figure 10 is an example of a point of variation with two alternative features: *Manual or Automatic*. The optional feature is a particular case of a variation point where the only possible choice is whether the property is present or not. The *Air-Conditioning* node represents another example of a variation point.

Furthermore, at the level of the features model, the variation points must be marked at all the levels of abstraction (requirements, architecture, implementation, testing, etc.). At the architecture level, Webber and Gomaa studied in their research [113] possible language extensions for the description of variability in product line architecture. Next section presents variability at the level of implementation.

### 2.5.3. Variability at the Level of the Implementation

There are a couple of techniques used at the implementation level which allow the management of variability. [99] and [3] are two reports which briefly present some techniques for the management of variability at the implementation level. Among them, here are five of the most useful already used successfully in the industry:

*The compilation techniques*. They allow the product derivation during the compilation phase. Conditional compilation and library loading are examples of these techniques. They are useful if variability is found at the level of code which needs to be included or excluded from the libraries they use.

*The programming techniques.* These techniques are used at the level of to the programming languages. The object-oriented languages (OOL) define useful methods to implement variability: inheritance mechanisms to describe abstraction, the method overloading and the dynamic linking. The variation points can be defined as abstract and then be implemented by variables in a given context. Some OOL make it possible to define generic classes, also called *template* classes. Thus, variability can be implemented by using the template classes when the alternatives are different only by the set of parameter types.

*The design patterns.* The design patterns [38] provide reusable solutions for specific type of problems. In [57], the Abstract Factory design pattern is used for the variable reification. The abstract factory makes it possible to define an interface for the creation of the concrete products. [63] presents a list of design patterns used to model variability in product families.

*The separation of concerns.* There are new approaches on software programming which can be used to implement and manage variability in the software systems. The separation of concerns [66] is an approach which helps to reduce the complexity of the systems or to manage complex tasks. It says that software should be decomposed in such a way that different "concerns" or aspects (a set of functional components) of the problem at hand are solved in well-separated modules or parts of the software. Many researchers [3] [11] [10] [44] propose to use this approach for the management of variability in SPL. Implementing variability in this way means to view the aspects as variation points and then each product, member of SPL, is differentiated by the set of aspects that it uses. [69] presents a case study on how to implement a product line using AspectJ.

*The generative programming* [26]. It is an approach which aims at creating software components, which, after suitable configuration, generate systems or other components. This allows building families of products (or product lines) out of which several concrete systems can be created. It is based on the concept of *generator*. Variability in SPL can be implemented by developing generators like generic artifacts and then their instantiation makes it possible to generate and re-generate sets of products. [9] proposes a methodology for generating product-lines (building variants of a program). [2] presents an approach to manage the variability based on the partial evaluation of the C language programs (declaring program specialization).

Even if these methodologies were successful at the implementation level and they are based and refined using technologies and techniques which also knew a relative success in developing software, nowadays the control of the code becomes more and more difficult to manage due to the exponential growth of the software complexity.

## 2.6. Constraints

In addition to variability, the software product lines are characterized by constraints which define the dependence between the variation points. In fact, the resolution of a variation point can influence the resolution of other ones. FODA, the domain analysis method previously presented, introduced rules for composition similar to the way constraints are used to describe dependences in a feature model. FODA [60] allows describing two types of composition rules: mutual-dependency (*requires*) and mutual-exclusion (mutex-with). The *required* rule indicates that one feature (optional or alternative) *requires* the existence of another feature (also optional or alternative) because they are interdependent. The *exclusion* rule indicates that one feature is *mutually exclusive with* another (they cannot coexist in the same product).

The product line constraints also appear at the architecture level. [67] considers that the constraints are part of the architecture of SPL. [56] presents a taxonomy of the variability dependences and constraints in SPL. This thesis also explored the works using UML for modelling the architectures of product lines to implement constraints of SPL (see section 2.3). One of the contributions of this thesis is to propose a way to integrate OCL (Object Constraint Language) [82] for the specification of these constraints in our approach of designing SPL; this is presented in chapter 4.

## 2.7. Conclusion

This chapter exposed the reality that the object-oriented approach does not provide all the solutions for software engineering today even if it represents a valuable basis for the description of further approaches. This remark can also be applied to component-based software engineering [100] and to the newer web services paradigm [46]. In particular, they do not provide a correct answer to the continuous evolution of the technologies: keeping applications up-to-date according to the evolution of technologies is too much time-consuming. It is also a lack of sufficient handling of clear designs and possibilities for reusable development of complex software using object-oriented programming. This is the context which explains the emergence of a whole set of new programming paradigms such as: Aspect Oriented Programming (AOP) [66], Subject Oriented Programming (SOP) [48], Intentional Programming (IP) [94], or Component Programming [100] [49].

This is the motivation for which today, the world of software engineering arrived to a paradigm shift from *object technology* to *model technology*, from *object composition* to *model transformation*. From objects and components, we can see other evolving trends like: processes, rules, services. Model technology is able to subsume most of these paradigms and others [59]. Therefore, "*model once, generate everywhere*" is the assertion that put a mark on the paradigm shift generated by the Model Driven Architecture (MDA)[86], but the road to model engineering takes time.

This chapter looked from the perspective of the paradigm shift that software engineering is experiencing today and introduces the state of the modelling software art. It shows the shortcomings of existing approaches and according to them it identifies and justifies features that a new approach should provide.

*Based on this motivation, this thesis promotes the idea that the most promising way to address software engineering is to provide an approach centred on models which captures the know-how of a domain, independently from both the software platform and from the possible applications.*

Next chapter presents SmartModels - an approach which relies on MDA and which is dedicated to contribute to strategies to increase software quality and productivity. Thanks to its meta-level, it clearly identifies the semantics of concepts used for the modelling of a given domain and on the other hand, thanks to approaches of separation of concerns and generative programming, it equips in a modular way the applications related to a particular target domain. The presentation of the key aspects when modelling in SmartModels is accompanied by the methodology to describe the structure and behaviour of a model and different casuistry which further explore and illustrate the flexibility and richness of the approach.

# 3. SmartModels – a Meta-Model Handling Generic Entities

Nowadays, companies involved in the development of modern software face several difficulties. One of the most important ones is the continuous evolution of software platforms (C++, Java, .Net, CORBA, EJB, Web Services, XML, etc.). One interesting solution to this problem is the Model Driven Architecture (MDA) approach from the OMG. It suggests that domain specific knowledge should be encapsulated in platform independent business models, apart from the applications. This solution is an answer to the failure of classical development techniques that rely on object-oriented design and programming.

The need for rapid evolution of the object oriented languages also involves the ability to adapt existing legacy software and to easily re-use it: to re-use classes or libraries of classes, models of the applications and even the know-how. Today many paradigms can be found which try to answer these problems. Here are a couple of examples: the separation of concerns paradigm, genericity, the model-oriented approaches, and the meta-modelling or programming using components. This thesis proposes an approach which relies on several of these paradigms in order to bring out to a solution for the re-use of the models of applications.

In most cases the description of a model means to forecast the various alternative values for an entity. In other words, certain entities of a business model are generic and this genericity must allow a structural as well as a behavioural variation of its properties in order to define the variation of a product line. This is the introduction of the genericity concept (the genericity traditionally groups the problems of variability and those of adaptation to a domain or a specific role) which will be developed by this thesis.

In addition, while being based on the ideas expressed by the separation of concerns paradigm, our approach encourages the definition of simple models (with a few number of entities): each one should address a limited set of problems corresponding to the product line or more generally to the business model. This is an important advantage because this consideration makes it possible to easily trace each entity and functionality and to increase the reusability.

One can think that this idea does not really help reusability due to the future difficulties at the time of the composition of these models: conflicts, renaming, adaptations and so on. But without a doubt, this approach includes a step that will describe a protocol of model composition which simplifies and automates clearly these tasks. This is not the primary focus of this thesis, but it constitutes an important aspect of the global solution which will be developed in the future [25] (see perspectives on section 7.3).

The result of privileging the simple, generic and easily reusable models is that it simplifies their reuse and makes it more effective.

According to these remarks, this thesis proposes another way to develop software: Model Oriented Programming (MOP) [106]. It is based on the Domain Driven Development (DDD) [27] track and introduces a macro level on top of the classical programming entities. It intends to be used for the handling, reuse and evolution of the business know-how and its associated applications. This chapter:

- It commences by presenting the Open Flexible Languages (OFL) [19], the starting point of the meta-modelling approach which intended to describe object-oriented programming languages based on classes;
- It introduces the concepts of the new paradigm MOP [106], which relies on MDA [86], aspect oriented and generative programming [26];
- It makes a comparison between MOP approach and component-based software engineering;
- It sets up two business model examples which will be further explored while developing the approach; and
- It presents the key aspects on modelling in our approach, called SmartModels, with respect to the level they manifest and based on customizable genericity.

## 3.1. OFL: Hyper-Genericity for Meta-Programming

In the interest of a better understanding of the SmartModels approach, it is important to get a glimpse of the starting point which is OFL: [16], [17], [18], [19]. SmartModels was born in the midst of OFL research and this can be observed in the introduction of important elements which SmartModels will deal later at another level of abstraction: concepts, atoms, parameters and characteristics.

OFL is the acronym for *Open Flexible Languages* and the name of a meta-model for object-oriented programming languages based on classes. OFL intends to describe languages especially by promoting capabilities such as introspection, modification and extension. OFL relies on three essential concepts of programming languages: the *descriptions* which are a generalisation of the notion of class, the *relationships* such as inheritance or aggregation and the *languages* themselves.

OFL provides a customisation of these three concepts in order to adapt their operational semantics to the programmer's needs. This section summarises the main characteristics of the OFL model, it shows how to create an application using this model and describes, as an example, the Java language according to OFL.

One of a project manager's main goals is to bring down the cost of software production. This mainly depends on two steps: programming and maintenance [109]. During these phases, the balance must be found between fastness and high quality. Several approaches are often used to solve this problem. Examples of some of these approaches can be given, but keeping in mind that none actually solve completely this problem at present:

- In a well determined context, such as the design of graphical interfaces or Web sites, the capacity to generate source code automatically brings valuable help.
- The efforts made to obtain more readable programming languages thanks to an ameliorated syntax contribute to improving the readability of the source code written in those languages.
- Reducing the gap between the design phase and the programming phase aims also to reduce the time spent in programming.
- Libraries of reusable components help not to start from scratch for each new piece of software.
- Design patterns offer model solutions used for specific programming problems.

- Aspect oriented programming addresses separation in terms of orthogonal services of an application's features, such as persistence or distribution of objects.

OFL deals with several of those solutions in a common approach starting with the idea that relationships between classes in object-oriented languages, and especially inheritance, are low-level mechanisms which it would be interesting to specify better. This approach is materialised in the definition of the OFL model [16].

OFL was first designed as a meta-object protocol such as than of CLOS [64]. However, more open and complete that CLOS, it has quickly become difficult and boring first to program and then to use it. So OFL switched to a hyper-generic approach [28] to solve this problem. Rather than allowing redefining behaviours by use of algorithms, it proposes a set of parameters. The algorithms, which are already implemented, take into account the values of these parameters to achieve the desired behaviour. These algorithms are called *actions* and they define the operational semantics. OFL promotes the idea that it is much more convenient for the meta-programmer (faster, more efficient and reliable) to set parameter values which drive well-tested actions, than to change the source code of several methods which describe altogether the semantics of the language.

The next sections will present the key-elements of the OFL approach. Briefly, this approach can be summed up as the search for a set of parameters whose values determine the operational semantics of an object language based on classes.

### 3.1.1. Hyper-Genericity

Genericity is the ability to customise the behaviour of a class in an object oriented language just as in the Eiffel [73] or C++ generic classes (template) [98]. Hyper-genericity is the ability to customise the behaviour of the language itself. More precisely OFL has chosen to customise the behaviours of three important notions of object languages based on classes:

- *relationships* such as generalisation and composition [80],
- *descriptions* which describe the application's objects, such as the classes and interfaces [4], [32], [43], and
- *languages* themselves.

**Parameters**. OFL defines a set of parameters [16] which represent the main features of the behaviours of these three important notions which are called *concept-relationship*, *concept-description* and *concept-language*. For instance, concerning the concept-relationship, the value of the *Cardinality* parameter allows specifying if it is simple or multiple. As for the concept-description we have for instance the *Generator* parameter which determines whether the concept-description can or cannot create its own instances.

**Actions**. The operational semantics of each concept must adapt to the value of its parameters. This is achieved thanks to a set of actions algorithms whose execution depends on these values.

For example, the assignment of an object to an attribute, the dynamic binding of the features, the sending of messages and many other behaviours are expressed according to parameters of concept-relationship and concept-description. OFL links two facets to each action: the first illustrates the static part inside an interpreter or a compiler; the second represents the dynamic aspect integrated within the runtime. The distribution of the code into these two facets depends on implementation choices of the OFL model.

### 3.1.2. OFL Objectives

The first main objective is *to improve the readability of the code* written in an object language based on classes. Indeed OFL allows specifying the relationships between the descriptions whose semantics are more precise than inheritance or aggregation. Since inheritance and aggregation are often used for very different purposes (for example: generalisation, specialisation, code reuse ...), OFL aims to offer the possibility to create a relationship which is specific to each of those uses. It is important to underline that in order to remain pragmatic, this approach does not aim to force the programmers to get out of their habits and to systematically interchange the relationships they are used to with the ones it proposes.

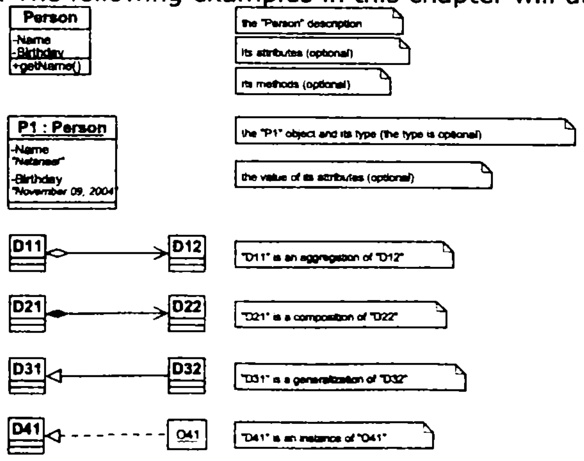
When a more specific relationship is used, readability of the code is improved (simplified). Furthermore, it will be easier to generate a relevant automatic documentation and the interpreter or compiler will be able to achieve more appropriate controls. As a consequence, it will be easier to maintain a program and to ensure further developments of the application.

OFL's second main objective is *to contribute to the reduction of the gap between the expressiveness of design methods and programming languages*. Indeed, one can be particularly pleased with a very suitable UML representation, but this is often difficult to implement straightforwardly using one's favourite programming language. OFL allows to define relationships with semantics closer to those which design methods and thus to program faster.

In order to obtain a realistic use of OFL, the programmer has to have access to libraries of concepts-relationships and concepts-descriptions from which he can select whatever he wishes to use. This method is similar to that which provides reusable software components [74].

### 3.1.3. OFL Model

Figure 12 presents the graphic conventions used by OFL approach and which are very similar to UML. The following examples in this chapter will utilize them.



Person
-Name
-Birthday
+getName()
the "Person" description
its attributes (optional)
its methods (optional)
P1 : Person
-Name
"Natanael"
-Birthday
"November 09, 2004"
the "P1" object and its type (the type is optional)
the value of its attributes (optional)
D11
D12
"D11" is an aggregation of "D12"
D21
D22
"D21" is a composition of "D22"
D31
D32
"D31" is a generalization of "D32"
D41
O41
"D41" is an instance of "O41"

**Figure 12. General Graphic Conventions**

### 3.1.4. General Architecture

Figure 13 illustrates how to use the OFL model to describe an application. This figure highlights three necessary levels for modelling:

- the application level which includes the program's descriptions and objects (*OFL-instances* and *OFL-data*),
- the language level which describes the components of the programming language (*OFL-components*), and
- the OFL level which represents the reification of those components (*OFL-concepts* and *OFL-atoms*).

**Application Level.** The programmer uses the services supplied by the language level in order to describe an application. He/she creates OFL-instances which are the descriptions and the relationships of his/her application by the instantiation of the OFL-components. At runtime, the application objects, called OFL-data, are instances of the OFL-instances representing the descriptions.

*OFL-Instances.* Each description or relationship described by the programmer is modelled by an OFL-instance. Figure 13 gives two examples of applications which each include five OFL-instances:

- three descriptions:
    - the Eclipse application contains: *EclipseProject*, *EclipseUML* and *UML_DiagramStudio*, or
    - the Photo-Cameras application contains: *PhotoCamera*, *DigitalCamera* and *Memory*
- one generalisation relationship:
    - EclipseUML inherits from EclipseProject, or
    - DigitalCamera inherits from PhotoCamera
- one aggregation relationship:
    - *EclipseUML* Project has an attribute of *UML_DiagramStudio* type, or
    - *DigitalCamera* has an attribute of *Memory* type.

*OFL-Data.* In an application each description instance is modelled at runtime by one or more OFL-data. Figure 13 shows some of them:

- *SmartModelsExtension* – an instance of the *EclipseUML* project description, and *SmartModels_Architecture* – an instance of the *UML_DiagramStudio* description
- *Sony* – an instance of the *DigitalCamera* description, and *SonyMemoryStick* – an instance of the *Memory* description.

It is important to point out that the OFL-instances which are descriptions specialise the OFL-atom object. Indeed, an object is the reification of the data of an application (OFL-data). So it represents the root of the specialisation tree of the OFL-instances which are descriptions.

**Language Level.** The language level describes different types of relationships and descriptions which can be used in the modelled language. The relationships are instances of concept-relationship and, the descriptions are instances of concept-description. The language itself is an instance of concept-language. Its main function is to put together the relationships and descriptions which are supplied to the programmer.

*OFL-Components.* The language level is solely composed of OFL-components. Figure 13 lists:

- several concepts-descriptions (e.g., *a-description*),
- several concepts-relationships (e.g., *a-generalisation-relationship* and *an-aggregation-relationship*), and
- a concept-language (*a-language*).

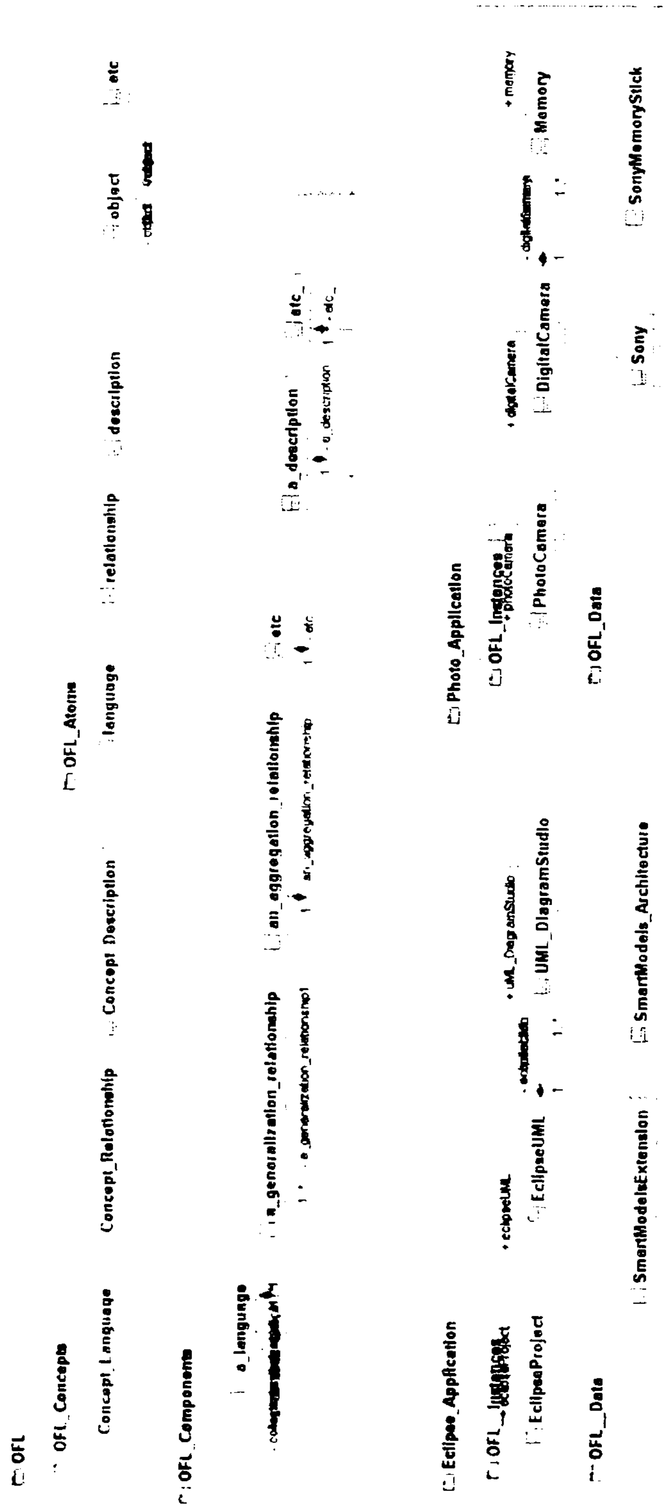


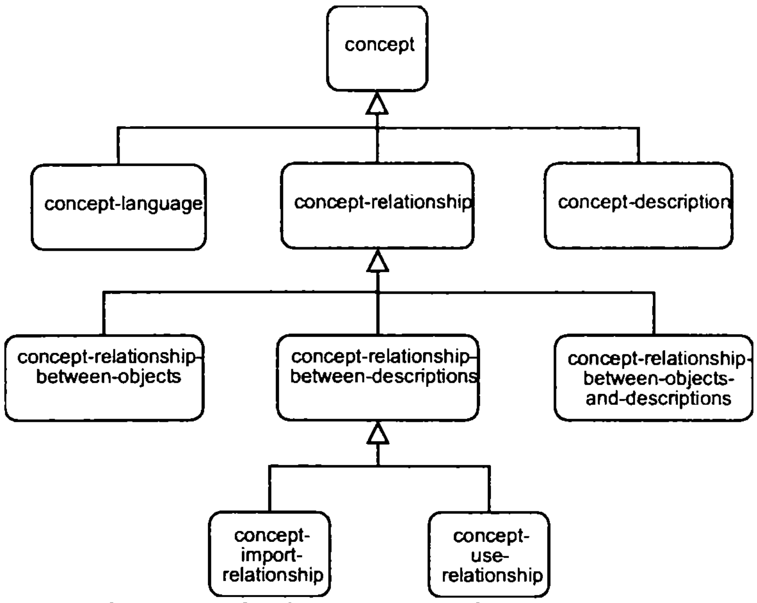
**Figure 13. The OFL Architecture**

It is possible to represent a concept-description as a meta-class and a concept-relationship as a meta-relationship and similarly a concept-language as a meta-language.

**OFL Level.** The OFL model is a meta-model for the programming language (language level) and as a result it is a meta-meta-model for the programs (application level). As already presented in section 3.1.1, OFL has chosen to customise three important notions: relationships, descriptions and languages. However, many other components need to be reified such as: objects, methods, assertions, etc., in order to model a language completely. The OFL level includes two types of entities:

- the OFL-concepts which describe the customisable part of the relationships, descriptions and languages, and
- the OFL-atoms which describe the non-customisable part of these three concepts as well as all the other components.

Also assertions are described in each OFL-concept and OFL-atom in order to keep the model consistent. For example, here is a structural constraint which is relevant to concept-descriptions and concept-relationships: let us assume that a description (an instance of a concept-description) has a relationship (an instance of concept-relationship) whose *Cardinality* parameter is set to 1 at this end. Therefore, an assertion has to verify every instance of this concept-description that it does not have other sources of relationships.


concept
concept-language
concept-relationship
concept-description
concept-relationship-between-objects
concept-relationship-between-descriptions
concept-relationship-between-objects-and-descriptions
concept-import-relationship
concept-use-relationship
**Figure 14. The OFL Concepts of Java language**

***OFL-Concepts***. Figure 14 presents the classification of the OFL-concepts. It is important to notice that in the OFL approach only the OFL-concepts are customised.

The meta-programmer's task is to create an OFL-component, i.e. an instance of an OFL-concept, by giving a value to each of its parameters. Thus he decides on the behaviour of each future instance of the OFL-component.

**Table 1. OFL concept-relationship parameters**

| Parameter | Definition | Type | Value Example |
|---|---|---|---|
| Name | It represents the name of the component-relationship. It must be unique within a language and cannot be redefined. | string | "Specialisation" |
| Cardinality | It expresses the cardinality of the target relationship using the syntax *1-n* which means that the relationships resulting from this component-relationship can be created between *1* description-source and *1* with ∞ description-targets (*n* represents ∞ or a positive integer greater than or equal to 1) | <integer, integer> | <1, ∞> |
| Polymorphism _policy | It is specific to the relationships of type *import* and it indicates if the polymorphism implied by the target relationship, applied both to methods and to attributes (this is why it is a pair of values) must be done according to a policy of redefinition (i.e., for Java attributes: *hiding*) (i.e., for Java methods: *overriding*). | <hiding \| overriding, hiding \| overriding> | <overriding, hiding> |
| Redefining | It indicates if the redefinition of the primitive types is mandatory, allowed or forbidden for the target relationship | <mandatory \| allowed \| forbidden, mandatory \| allowed \| forbidden, mandatory \| allowed \| forbidden, mandatory \| allowed \| forbidden> | <allowed, forbidden, allowed, forbidden> |
| Sharing_level | It is specific to the relationships of type *use* and it indicates if the instance of the description-target can be shared | global \| package \| description \| instance \| unique_instance | instance |

If the operational semantics which the meta-programmer wants to bind to an OFL-component does not match the planned actions, then he has to modify the code of those actions. The OFL model is left open by this possibility which should not be often used, except in a very specific context. Indeed, in that case, the meta-programmer's job is much heavier than just giving values to parameters.

*Concepts-Relationships*. A concept-relationship is the entity which represents a kind of relationship. A concept-relationship is consequently a meta-relationship. Among the relationships which are to be found in many object-oriented languages based on classes and object design methods, OFL applied to an object-oriented programming language may mention, for example: inheritance, aggregation, composition, generalisation, etc. However a given design method or language seldom uses all of these relationships and usually contains some of them in order to simulate others. For example the generalisation in UML describes a generalisation as well as an inheritance, a strict sub-typing: these three relationships have different semantics even if they are similar enough to be often confused.

Around thirty parameters define the semantics of the entire OFL model's concept-relationships. Table 1 lists some examples.

Figure 16 lists the concepts-relationships representing the relationships of the Java language (this is one of the main two examples which the next section will address and develop while presenting the SmartModels approach).

Figure 14 illustrates our classification of the concepts-relationships. Concerning the inter-description relationships, we distinguish between the *import* relationships (generalisation of the inheritance mechanism) and the *use* relationships (generalisation of the aggregation mechanism). As for Figure 13, it illustrates one instance of an import concept-relationship (*a-generalisation-relationship*) and one example of an use concept-relationship (*an-aggregation-relationship*).

OFL also takes into account the relationship between objects and classes which is used for example to model the instantiation relationship existing between an object and its class. It is also possible to model the relationship between objects, although OFL mainly concerns the inter-description relationships.

*Concepts-Descriptions*. A concept-description allows defining the notion of class and all that looks like a class such as the interfaces in Java. Therefore a concept-description is a kind of meta-class. For instance, we can notice that, even if they look the same, the Eiffel, C++ or Java classes, they show some notable differences. Figure 13 gives one instance of concept-description called *a-description* as an example.

Around twenty parameters are necessary to describe the behaviour of a description in the OFL model and Table 2 lists some examples. Each concept-description is compatible with a set of concepts-relationships. For instance, in Java, the concept-description *Interface* is compatible with the concept-relationship *Implementation*, but it is incompatible with *between-classes-inheritance*. Figure 13 presents also the example of photo-cameras. In this case the concept-description *DigitalCamera* is compatible with the concept-relationships *CaptureLight and SaveOnDigitalMemory*, but it is incompatible with *SaveOnPhotographicFilm*.

*Concepts-Languages*. The concept-language is an important and yet simple notion. It models a language. In particular, each language includes a set of concepts-descriptions and a set of concept-relationships which are compatible with at least one of the concept-descriptions. In Figure 13, there is only one concept-language's instance (*a-language*) which represents the modelled language.

**Table 2. OFL concept description parameters**

| Parameter | Definition | Type | Value Example |
|---|---|---|---|
| Genericity | It specifies if the target description is generic, *true,* (it can represent a set of types) or not, *false.* | boolean | false |
| Generator | It specifies if the target description is a generator (it can create instances). | boolean | true |
| Visibility | It indicates if the target description is visible by a set of given entities. | global \| package \| description \| method \| object \| statement \| expression \| list | global |
| Attribute | It specifies if the target description can define attributes, *allowed* or not, *forbidden*. | allowed \| forbidden | allowed |

The concepts-languages are not much customised. Their main function is to federate the concepts-relationships and concepts-descriptions which are compatible with them.

***OFL-Atoms***. They represent the reification of the non-customised entities of the model. Figure 15 illustrates a set of those OFL-atoms. The relationships, descriptions and languages have their own OFL-atoms to describe the part of their structure and their behaviour which are not customised. For instance, Figure 13 highlights that the OFL-component called *an-aggregation-relationship* is a specialisation of the OFL-atom *relationship*.

Also, in an application all the features of a description are instances of an heir of *feature*, all the expressions are instances of *expression* or of one of its heirs and all the objects are instances of *object*. Thus OFL gives a full reification of the entities found in an application at runtime.

## 3.2. The Model-Oriented Programming Approach

The object-oriented approach does not provide all the solutions even if it represents a valuable basis for the description of further approaches. This remark can also be applied to component-based software engineering paradigm. In particular, they do not provide a correct answer to the continuous evolution of the technologies: keeping applications up-to-date according to the evolution of technologies is too much time-consuming. This thesis promotes the idea that to provide an approach centred on models which capture the know-how, independently from both the software platform and the possible applications, is promising.

This section introduces the Model Oriented Programming (MOP) framework with a set of essential entities. They are a first attempt for the definition of the main principles of the approach. A second contribution is the proposal of SmartModels which is one interpretation of these principles. The next section will draw a parallel between this approach and component technology.

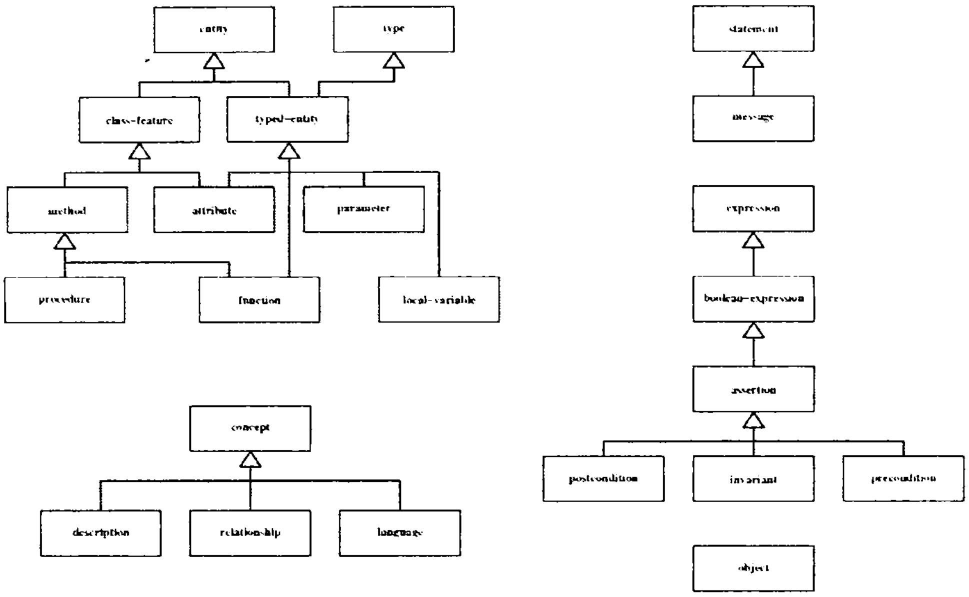


**Figure 15. OFL-Atoms of a Modelled Language**

MOP moves the accent from objects and components to the more dynamic perspective: the models. Our approach on MOP is original for software engineering because:

- It aims to provide a framework for describing models in which to encapsulate the specific knowledge of a domain according to multi-system scope development defined in domain engineering [26];
- It ensures a clear separation between the model and the technologies which make the model executable by a software platform;
- It incorporates software factories (SmartFactory) which automate as much as possible the code generation and also provides easy and clear entry points in the code for the programmer to change or update the code;
- It integrates new ideas from AOP [66], SOP [48], IP [94], web services [46] and component-based engineering [100] to build flexible, readable and easy maintainable software;
- It relies on W3C (*World Wide Web Consortium*) [46] and OMG (*Object Management Group*) [85] standards like XML and DTD for serialization of the models, OCL [82] for assertions, MOF [84] and AST [7] for the meta-model [24] description;
- It captures the semantics of a model and proposes a way to handle their treatment.

In order to provide the facilities presented above MOP deals with a set of entities. Many of them are familiar concepts from object-oriented programming and the next sections will present them together with their first validation in the context or the interpretation of our approach called SmartModels.

The kernel of SmartModels is represented by a Meta-Object Protocol (*mop*) and the abstract reification of the entities of the meta-model. Right now it consists in a set of Java classes generated and customized with the Eclipse EMF plug-in (see section 3.4.2 for the motivation of this choice) which encapsulates the basic rules and the definition of the entity types of the approach [84]. Its architecture was driven by previous works on OFL [24] and it is formed by:

- a class (in fact the so-called *mop*) which retains the place in the structure of nodes of each entity of a business-model (its name, the name of the super-class specialized by this entity and the name of the meta-class - the entity that provides the meta-information) and its extent (the collection of its instances). This collection is updated automatically when a new instance is created and through the previous information it is possible to implement a management on the list of instances of entities (to access and update also the *mop* of the parent of this entity and of its meta-description);
- a set of classes which provides basic tools for managing the entities: the definition of a collection (with some optional facilities: to be ordered, to accept only distinct objects, to set a minimal or maximal cardinality), OFL basic types (Enumeration, Tuple, Integer, Boolean, ...), the definition of an OFL value, types of redefinitions for parameters and characteristics and OFL constants;
- a set of classes which define the abstract core representation of each entity of a business-model.

As a consequence of the fact that any entity of the kernel of SmartModels (which is built-in) is a *"first-class"* entity and it may be specialized through inheritance, therefore the kernel can be bootstrapped and so it can automatically generate itself. Thus it represents an open and flexible platform to describe business-models. Any further changes in the kernel will be seen by all entities.

### 3.2.1. *mop* versus Components

SmartModels and *mop* offer practical solutions to several important problems found in object-oriented and component-oriented programming. This section aims to draw a parallel between this approach and component technology. Later, this thesis will draw the same parallel addressing implementation issues of the validation prototype - SmartFactory.

In the previous decade most of the articles published on these matters seem to point to components [100] as the solution to a new maturity level of the software engineering. Building new applications by combining ready-made (bought) and custom-made components improves quality and supports rapid development because it enables the reuse of software. Component-based software engineering approaches plead for building reusable components that can be plugged together to create new systems.

First of all, finding the equilibrium it is again a hard thing: it is a decision between generality and specialization. General components can be used in many applications, but may be much more bloat artefacts than needed. Specific components, on the other hand, are appropriate for a given situation, but then there may be many specific components that the clients have to search through to find the one that exactly meets their needs. Some of the components may need to be adapted to the particular situation and this can be another problem.

In this context, *mop* offers the possibility to organize all the entities defined for a business-model in a hierarchy. In this way a model can have at the same time components more general and more specific having the possibility to use or not the advantages of inheritance for meta-information. At any time a component can be specialized to be adapted for a certain situation using the inheritance. Also, a more specific component can use or not the same semantics as its parents, or it can have different values for its meta-information.

Other problems in software engineering are linked with the systematic interplay of components. It would be naïve to assume that we can simply select components from a well organized repertoire and after a click we have the final application. In reality many questions arise on how the abstract interaction of components can be described, how variety and flexibility can be covered in the design of a component, how critical system properties can be guaranteed when there are many vendors or how can performance be guaranteed.

*mop* handles variabilities and commonalities through meta-information (see section 3.4.2). The behaviour of an atom is influenced by the values of its parameters and characteristics and both, the actions defined in its meta-level (concepts) and the set of assertions attached to each atom, will check the conformity and support aspects [66]. All the components provided by different vendors will have to conform to the meta-information from the business-model.

All current software component camps are not on a domain-specific standard. Components-based reuse has proved useful in some application domains. For example, libraries of mathematical functions are commonly used. However, in other domains, component-based approach poses problems that obstruct effective reuse. Notably, methods for searching, analysing and customising components and integration of components into a working system are not well defined, explained and understood. SmartModels approach based on *mop* follows the Domain-Driven Development [27] principles and therefore offers a framework for developing domain specific applications. In this context the concern is to define an approach which makes possible to specify any model applied to any domain: object-oriented languages, digital cameras, nuclear factory which produce electricity, etc.

Proliferation of new component technologies is another issue. To obtain a component-based application, a developer must choose between at least three component technologies: CCM (CORBA Component Model), EJB (Enterprise Java Bean) or Web Services. In SmartModels the technologies are defined independently from the approach. They contain functionalities which allow defining more easily the application. For example, the DOM API is welcomed to manipulate XML representation of the business models. Other information may be added in order to generate visit entities (see section 3.4.4) which fit exactly the expectations of the programmer. Indeed the source code generation is essential because it allows him to focus only on the visit entities which are addressed by the facets and to be assisted for the description of their behaviour.

## 3.3. Introduction to SmartModels Meta-Model

A very promising approach towards a better and more practical framework for software development is the vision of OMG: Model-Driven Architecture (MDA) [86]. It proposes to separate Platform-Independent Models (PIM) from the perspective of Platform-Specific design and implementation Models (PSM). Due to the rapid platform evolution (the arrival rate of new technologies is increasing and this rate is not likely to decrease in the future) and the growth of the software systems complexity, the model becomes the most important concept rather than the code or other implementation issues.

That is why we are facing today a shift in software engineering between object technology and model technology. Model engineering tries to solve more than only the platform independence problem, but it also moves the accent from objects and components to more dynamic and evolving perspectives: models, processes, activities and services.

Taking into consideration the new MDA paradigm, generative programming and other trends in current research communities for software engineering, this thesis proposes a new and challenging view on developing software, SmartModels, an approach based on Model-Oriented Programming ([105] and [106]).

Historically, my interest for modelling and generic parameterization started from studying [16] and with the objective of modelling the object-oriented languages, thanks to a meta-model, using a meta-object protocol (*mop*) and by introducing the concept of hyper-generic parameter.

The '*hyper*' prefix stands for the idea that this type of parameters controls all the behaviour of the system which is modelled. In section 3.4, as the approach is developed, the prefix will probably not be used so often in order to simplify the text, but every time the term 'parameter' will pop-up, it will have the same connotation: 'hyper-generic parameter'.

Open Flexible Language (OFL) is a system formed of a meta-model and a meta-object protocol (*mop*). Through entities at the meta-level and generic parameters, OFL offers a technique for representing, modelling and adapting an object-oriented language.

For example, it identifies the following meta-entities: meta-class (traditional concept), meta-relationship (i.e., for modelling the behaviour of inheritance and aggregation relationships), and meta-language (to join meta-classes and meta-relationships). These terms have been adapted for this section and it is important to note that the discussion is at the level of a language and not of an application.

As an illustration, this means that the term '*class*' represents the notion of a class and that a '*meta-class*' is an entity able to generate various types of classes (and not just classes). Therefore, related to the application level, this notion is a meta-meta-class for the instances of the generic classes.

Each one of these entities is defined through generic parameters. For example, a meta-class has a '*Creator*' Boolean parameter which indicates if the class can or cannot create its own instances. Thus, in the context of the Java language example, '*Class*' has value '*true*' for this '*Creator*' parameter and for the 'Interface' is '*false*'. This parameter controls the behaviour of these entities.

Having the OFL experience it became interesting to add new characteristics which were not previously predicted. Therefore, SmartModels incorporated work on separation and composition of concerns [106]. The new ideas and concepts which resulted from these approaches brought important contribution on modelling of the applications and, in particular, of the business models. It is thus on this level that the meta-modelling techniques with generic parameters are applied in our approach [25] in order to increase the expressivity of the models and in the future to use adapters to compose these models [68].

SmartModels tries to be part of the new trends in current research communities for software engineering. It proposes another way of developing software: *Model Oriented Programming* (MOP). It is based on the Domain Driven Development track (DDD) [27], which relies on several paradigms such as object-oriented technology, languages for components, MDE, approaches for the separation of concerns, and generative programming [26].

This proposal relies on previous works which deal on the one hand with meta-modelling [24], and on the other hand with the design of a software factory called SmartTools [7]. It intends to enrich both approaches in order to make easier the development of domain specific applications.

### 3.3.1. Criteria for evaluation and objectives of the approach

Based on the conclusions from the state of the art (see section 2.7) and OFL previous experience, Table 3 addresses the essential criteria used to evaluate the current approaches on developing lines of software entities which was also used as the main decisive factor in defining the new approach:

**Table 3. Criteria to evaluate meta-modelling approaches**

| CRITERIA | DESCRIPTION |
|---|---|
| Integration of MDE principles | To create means for developing reusable models: methodologies and available modelling technologies, tools infrastructures (frameworks), PIM and PSM modelling, transformation of models |
| MOF+UML | To offer flexible support for features widely used by tools based on MOF and UML, analyze the capability of the approach to abstract required aspects of the object systems into models and to keep them consistent. |
| Integration of AOP | To make available cross-cutting features which can be used to get additional views of the system (handling complexity of a system allowing decompositions according to aspects: trace, error handling, new functionalities). |
| Handling commonalities and variability | To provide methods to identify commonalities among members of a line of products (specifications, modelling the domain, reuse of generic product in the family), mechanisms to handle the variabilities (functional and non-functional requirements, different characteristics, platforms, unexpected variants) - this means constraint based scheduling and providing more guidance in context. |
| Automatic Code Generation | To use platform independent meta-models and to map them on specific platform models: the goal is to provide quality and as much as possible quantity code generators (effective and automatic generation of large portions of applications) and execution infrastructures - this means automation through tools, patterns, framework, templates. It also makes use of the domain specific languages to solve targeted problems and to enable efficient use of the model driven design approach - using the Domain Specific programming Languages (DSLs) tools. |
| Application development | To analyze capabilities for rapid building and deploying applications related to the model |

To sum up the criteria for evaluation, here are the main objectives of SmartModels:

- on the one hand, to clearly identify, thanks to a meta-level, the semantics of concepts used for the modelling of a given domain, and
- on the other hand, thanks to approaches of separation of concerns and generative programming, to equip, in a modular way, the applications related to this domain.

SmartModels is a set of domain specific models dedicated to the development of software. This approach is original and may be distinguished from other approaches by the following characteristics:

- it introduces on top of the entities which structure the model (reification level), a semantic layer which enables to define and factorize the basic functionalities related to the domain,
- it provides a set of facilities (in order to quickly build applications related to the model), which strongly rely on the two levels of the model (data and semantic models),
- it ensures a clear separation between the model and the technologies which make the model executable by a software platform.

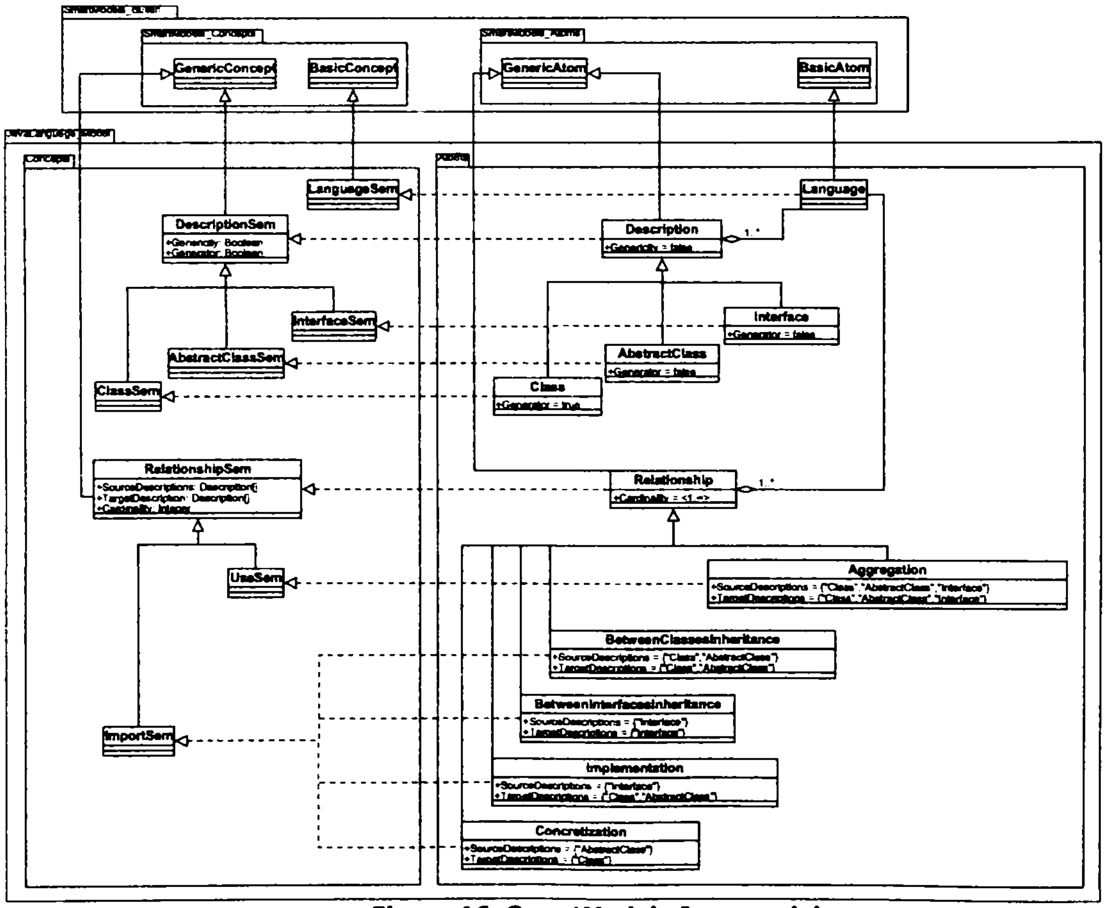

GenericConcept
BasicConcept
GenericAtom
BasicAtom
LanguageSem
Language
DescriptionSem
+Genericity: Boolean
+Generator: Boolean
Description
+Genericity = false
1..*
InterfaceSem
Interface
+Generator = false
AbstractClassSem
AbstractClass
+Generator = false
ClassSem
Class
+Generator = true
RelationshipSem
+SourceDescriptions: Description[]
+TargetDescription: Description[]
+Cardinality: Integer
Relationship
+Cardinality = <1..*>
1..*
UseSem
Aggregation
+SourceDescriptions = {"Class","AbstractClass","Interface"}
+TargetDescriptions = {"Class","AbstractClass","Interface"}
BetweenClassesInheritance
+SourceDescriptions = {"Class","AbstractClass"}
+TargetDescriptions = {"Class","AbstractClass"}
BetweenInterfacesInheritance
+SourceDescriptions = {"Interface"}
+TargetDescriptions = {"Interface"}
ImportSem
Implementation
+SourceDescriptions = {"Interface"}
+TargetDescriptions = {"Class","AbstractClass"}
Concretization
+SourceDescriptions = {"AbstractClass"}
+TargetDescriptions = {"Class"}

**Figure 16. SmartModels Java model**

The main interest of such an approach is to provide the power to define the semantics of the entities which are addressed by a model, independently from any application. In general, the semantics are spread out in the applications which may directly handle the model.

SmartModels does not make any difference between the modelling of the business model and the modelling of its applications. Thanks to the semantics which is encapsulated in the entities, related applications may handle directly this knowledge without going through some implementation phases (the generation process takes care of this).

It is very important to know that contributions of both generative programming and separation of concerns are used in order to achieve a better flexibility and modularity of the applications related to the model.

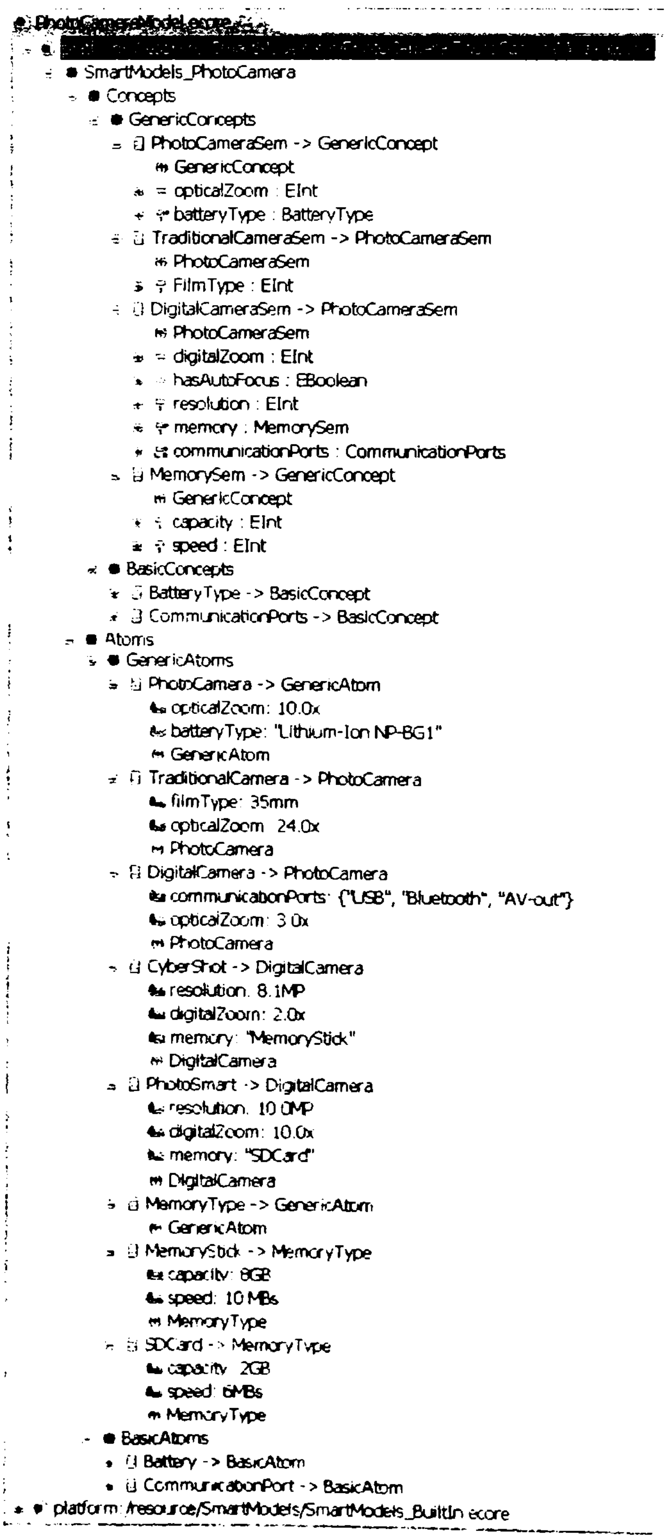
This section explained the various stages which led to the proposal of a meta-model for the specification of business models and next sections will describe its main aspects applied to two possible example models. Next chapter will show that this approach, which is based on customizable genericity, provides the needed expressivity for modelling product lines. Then it follows a chapter which describes the key elements of an Eclipse plug-in [35] for the implementation of the approach. Finally, the last chapter evaluates this work comparing to the state of the art and then wrap up by giving future perspectives.

## 3.4. Key-aspects of modelling in SmartModels

This section introduces the main elements that can be used to describe a business-model in SmartModels. For a brief presentation see [106]. A business-model is defined through the identification of its entities according to the know-how of a specific domain. This process consists in producing an XML document (i.e. by a parser of the domain specific language) compliant with the AST (or DTD) which describes a model in our approach.

This document will drive the generation process of a class (a Java class in the current version) for each entity. Then this set of generated classes, considered an implementation of the business-model, is attached to *mop* as sub-hierarchies of the built-in kernel (adding also features for handling access in the specialized and meta-hierarchies and its extension and for loading/saving instances of entities from/into XML streams).

Next section introduces the examples through which the presentation of SmartModels entities will be explained. Then sections 3.4.2 to 3.4.4 present each entity with respect to the level it manifests. Figure 18 distinguishes between the different levels of the architecture of our meta-model: the key aspects proposed by SmartModels in order to define business models. They will be used by generators in SmartFactory in order to produce code attached to the *mop*. Let us follow them from top to bottom level.



```
PhotoCameraModel.ecore
  [illegible]
    SmartModels_PhotoCamera
      Concepts
        GenericConcepts
          PhotoCameraSem -> GenericConcept
              GenericConcept
              opticalZoom : EInt
              batteryType : BatteryType
          TraditionalCameraSem -> PhotoCameraSem
              PhotoCameraSem
              FilmType : EInt
          DigitalCameraSem -> PhotoCameraSem
              PhotoCameraSem
              digitalZoom : EInt
              hasAutoFocus : EBoolean
              resolution : EInt
              memory : MemorySem
              communicationPorts : CommunicationPorts
          MemorySem -> GenericConcept
              GenericConcept
              capacity : EInt
              speed : EInt
        BasicConcepts
          BatteryType -> BasicConcept
          CommunicationPorts -> BasicConcept
      Atoms
        GenericAtoms
          PhotoCamera -> GenericAtom
              opticalZoom: 10.0x
              batteryType: "Lithium-Ion NP-BG1"
              GenericAtom
          TraditionalCamera -> PhotoCamera
              filmType: 35mm
              opticalZoom 24.0x
              PhotoCamera
          DigitalCamera -> PhotoCamera
              communicationPorts: {"USB", "Bluetooth", "AV-out"}
              opticalZoom: 3.0x
              PhotoCamera
          CyberShot -> DigitalCamera
              resolution. 8.1MP
              digitalZoom: 2.0x
              memory: "MemoryStick"
              DigitalCamera
          PhotoSmart -> DigitalCamera
              resolution. 10.0MP
              digitalZoom: 10.0x
              memory: "SDCard"
              DigitalCamera
          MemoryType -> GenericAtom
              GenericAtom
          MemoryStick -> MemoryType
              capacity: 8GB
              speed: 10 MBs
              MemoryType
          SDCard -> MemoryType
              capacity 2GB
              speed: 6MBs
              MemoryType
        BasicAtoms
          Battery -> BasicAtom
          CommunicationPort -> BasicAtom
  platform:/resource/SmartModels/SmartModels_BuiltIn.ecore
```

**Figure 17. SmartModels photo camera model**

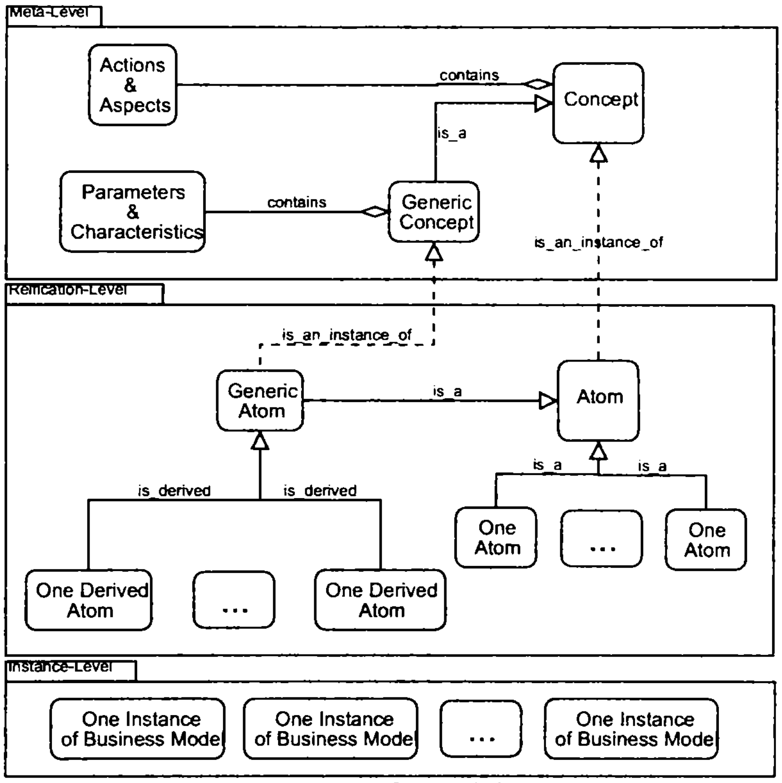


**Figure 18. Key-Aspects of a Model in SmartModels**

### 3.4.1. Casuistry

This section sets up two business model examples which will be further explored in the next sections while developing the approach. Chapter 6 presents a complete implementation of one of them.

First example is more abstract and carries on the interest of OFL approach (see previous section) to create a meta-model for object-oriented programming languages based on classes with an application to Java: how to design their entities (like notions of class or interface, relationships like generalization or composition), how to customize and extend them (i.e., the polymorphism), how to architect assertions to control their behaviour (like visibility issues, method redefinition rules), how to create applications.

The second example models a part of the domain of photo cameras. The goal is to develop a more concrete illustration on how to use SmartModels for modelling the entities which differentiate each product in this context: how to architect diverse types of photo cameras (traditional or digital), how to describe their functionalities (basic and optional features), how to separate concerns (like the type of supply source, the supported memory sticks, the built-in communication ports, the zoom effect provided by the cameras lenses, the resolution captured, the installed software firm like the algorithm used for compression).

Next sections will further investigate these examples at the moment of the introduction of each SmartModels entity.

## 3.4.2. The meta-level

First of all, the *meta-level* is the top level of SmartModels business-model reification and it handles the meta-information through concepts. A concept participates to the definition and the management of the meta-information of a business-model. It encapsulates the semantics of entities and their treatments. It can be related to one or a number of atoms (see section 3.4.3) and drives their behaviour (in SmartModels approach an atom is the structure which encapsulates the description of an entity).

Figure 16 illustrates the SmartModels model of the Java programming language. It identifies only one basic concept – the *LanguageSem*, because it does not raise any interest for now to add more semantic information for the language itself. On the other hand, the diagram distinguishes two hierarchies of generic concepts:

- *DescriptionSem* which is the abstract concept for all Java descriptions, like *class, abstract class, interface*. They are also represented in the diagram due to the fact that each one of these entities is motivating to extract more specific semantics.
- *RelationshipSem* which is the abstract concept for all Java relationships between description entities. The hierarchy presents *import* and *use* type of relationships [23].

This is not an exhaustive list of Java language concepts, but the choice was to have an easily understandable diagram with the most representative entities.

A concept makes the clear distinction between the semantics (meta-level) of the entities of a business-model and their reification (reification-level). As a result there are several very positive consequences:

- the maintenance of the semantics (updating and redefining of the semantics) deals only with concepts;
- the support for reuse of the semantics in other (closely related) business models; and
- the model transformation which is one of the key points of this approach.

At this point it is important to mention that the elements that compose the concepts address only the semantics of the entities and not their instances (semantics of a business-model are independent from an application). Also, the concepts are organized in a hierarchy so we can take advantage of the polymorphism to reuse semantics for atoms from the same reification.

Figure 17 presents the example of a SmartModels photo camera model and the following basic concepts can be observed:

- *BatteryType* and *CommunicationPorts*: none of them are represented in detail. If there is not interest in additional semantic information an entity is considered *basic* (non-generic)

The list of generic concepts is developed more and here is an examination of the example:

- an abstract *PhotoCameraSem* concept which encapsulates the common semantics of all photo camera represented in this model;
- *TraditionalCameraSem* concept which contains the semantics to handle traditional photo cameras and *DigitalCameraSem* concept which includes the common semantics for all the hierarchy of digital cameras supported by this model.

Therefore, the generic concepts hierarchy profit from polymorphism and the following paragraphs will explain the way they encapsulate the semantics.

A meta-model makes it possible to define the level of genericity for each entity when describing a business model. The generic part expresses the variability and it is possible to derive entities by simple instantiation. The basic elements needed for the specification of a business model (attributes, associations, methods, classes, basic types, etc.) are present in the existing meta-models such as MOF, UML or EMF. They all have the capacity to bootstrap (to describe themselves), too.

The SmartModels meta-model, which is developed in this thesis, is an extension of EMF and the motivation of this choice is the following:

- EMF is of a big interest its current releases and in the same time it is sufficiently expressive to describe a business model, and
- EMF was developed and it is part of the Eclipse platform which opens excellent opportunities for the construction of a prototype.

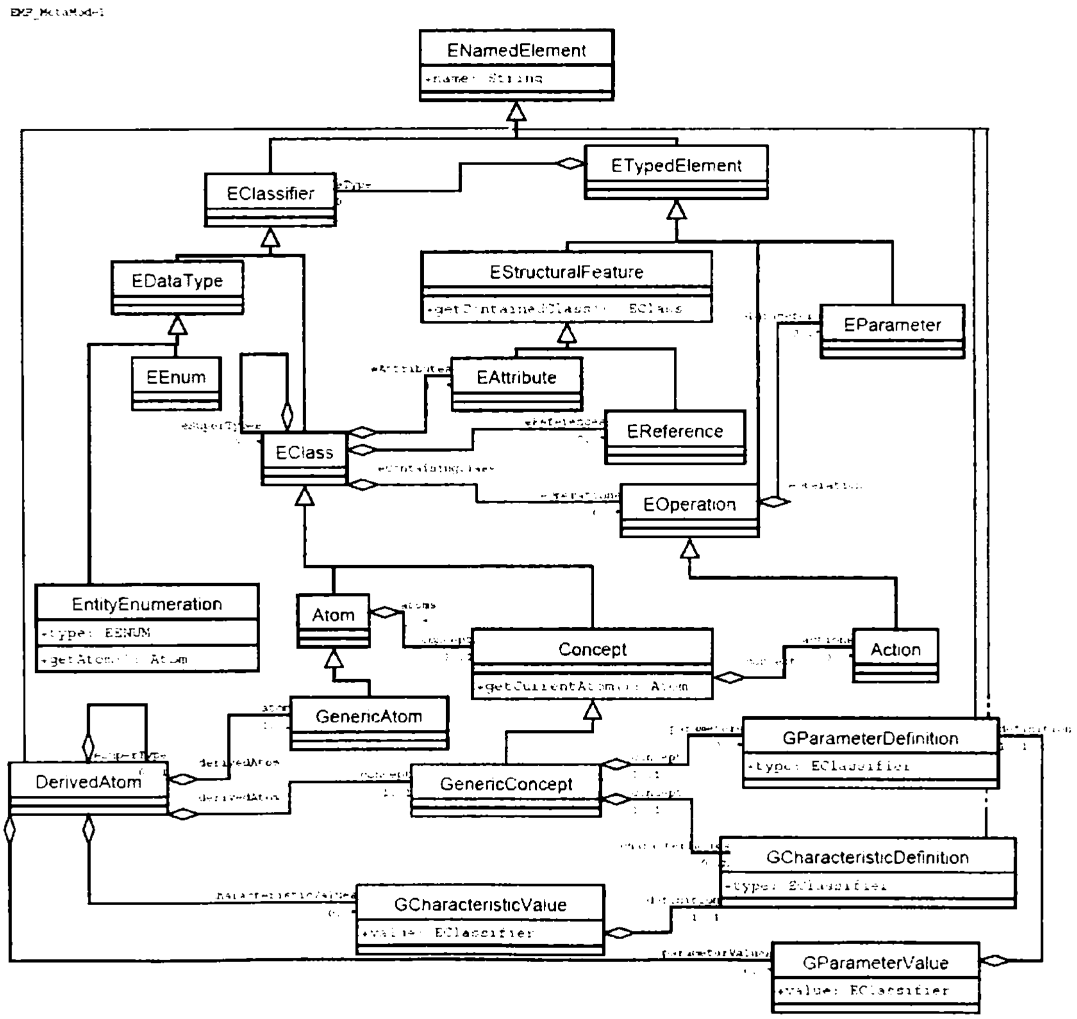


**Figure 19. The Basic Architecture of SmartModels as an Extension of Eclipse EMF**

The SmartModels entities and the models which it describes are equipped with assertions implemented with an extension of EMF for the description of constraints in OCL [82], [114].

Figure 19 presents a sub-set of SmartModels meta-model described using Eclipse EMF plug-in (some of the entity names were willingly shortened to save space) with the main entities of the EMF model on which the extension is based. The entities which constitute Eclipse EMF model are presented in the large EMF_MetaModel rectangle.

It is important to highlight the ENamedElement root which guarantees that each entity has a name (this is a must for all first-class SmartModels entities). EClassifier is the entity which represents an equivalent of the UML classifier or an OFL description - this means everything that it is similar to the class notion (a container of objects) or a package (a container of classes or other containers). EAttribute and EOperation reify the concepts of attribute and method. All the other entities which do not belong to EMF_MetaModel are part of SmartModels extension and will be presented in the following section.

The semantics of a business-model stored in a concept are reified through a set of **hyper-generic parameters and characteristics** [24] (which form the meta-information) and a set of **actions** (which perform treatments on the entities according to their meta-information). The identification of the parameters and characteristics and their possible values is the job of the meta-programmer which addresses the know-how of the domain.

The hyper-generic parameters customize the behaviour of the entities of a business-model. They refer to generic atoms (see section 3.4.3) and not their instances. Their role is to capture and express the properties which compound the definition of the generic entities. A *parameter* expresses a basic type property, e.g. a *Boolean* or an *Integer* value, an *Enumeration*, a *Tuple* type or a *Collection* of values. A *characteristic* expresses a property whose value is defined by an atom or a set of atoms (*Enumeration*, *Tuple* or *Collection*). In order to describe the behaviour of a generic entity the programmer has to set those values. For example, a business-model built to encapsulate the structures and semantics of an object-oriented programming language may define *parameters* like:

- *Cardinality* which expresses if there is simple or multiple inheritance - Java simple inheritance set the value to 1.
- *Generator* which specifies if the given entity can create or not its own instances:
    - "Class" notion in Java can generate instances - value is "*true*" because it can provide the possibility to create a constructor or redefine the default one;
    - "AbstractClass" and "Interface" notions cannot create instances, therefore the value is "*false*".
- *Genericity* - also a boolean parameter which specifies if the given entity is generic, if it can be a template that represents a set of types. It is one of the most anticipated and debated enhancements to the Java language in 1.5 version release. The parameterized types were formally proposed through Java RFP by the time my doctoral program started;

or characteristics like:

- the collection of valid kinds of classifiers for a given type of inheritance:
    - the valid *source descriptions* for the first end of the association and
    - the valid *target descriptions* for the second end of the association.

Besides the example of modelling a programming language which is more abstract, lets further explore the photo camera model to identify possible semantics. Here is the list of parameters according to their concepts:

- *opticalZoom* and *digitalZoom* which indicate the maximum zooming capability of a derived product. They are of type integer and the optical zoom can be found in all types of cameras, whether the digital zoom is just part of the digital camera semantics;
- *filmType* identifies the code of the film category supported by this family of traditional cameras (i.e., 35mm, 20mm);
- *hasAutoFocus* specifies if the camera is equipped with sensors to determine and adjust the optical system to correct focus (value is *true*) or not (*false*). This parameter is conditioned by a positive value for optical or digital zoom parameter;
- *resolution* determines the limit of resolution supported by the lenses of the family of cameras modelled by this concept (i.e., 8.1MP, 10.0MP);
- *capacity* and *speed* are parameters which express the boundary values for the major features of a digital camera memory stick.

All these parameters illustrate properties of camera entities which are expressed using basic types (i.e., integer, boolean). Characteristics address the set of properties whose descriptions deal with one or a set of other entities in the model (atoms from the model, i.e., *BatteryType* – see section 3.4.3). The photo camera model from Figure 17 enumerates the following characteristics:

- *batteryType* which specifies what kind of batteries each category of cameras support. The target of this characteristic is instances of *BatteryTypeSem* basic concept;
- *communicationPorts* which is similar but applied to communication ports supported by the family of digital cameras. However, the cameras semantics model assumes that there is only one type of battery suitable to a camera, while it can incorporate a collection of supported ports;
- *memory* which specifies the compatible family of memory sticks for a given family of digital cameras.

From these examples the following question can be raised: how will the semantics link and control the behaviour of their instances? The relationship between the semantics and their reifications will be presented in section 3.4.3, but next paragraph addresses the key entity which is important to control a model created by a programmer according to the meta-information: *actions*.

Actions are *first-class* entities addressed by concepts in order to dynamically manage the behaviour of atoms according to their meta-information. The body of an action encapsulates the execution which can be performed by that action. The execution of an action depends on:

- querying the parameters and characteristics of the generic atom to which the action is attached;
- a set of invariants, pre-conditions and post-conditions;
- an optional set of aspects;
- additional information provided by the meta-programmer.

An action must be completely independent from the application related to the business model. Therefore a typical scenario is that the behaviour of a given application relies on the semantic model, that is to say call those actions or query hyper-generic parameters.

Further investigation on the above examples help to imagine needed actions in order to verify constraints on the model entities. Here are some Java language model actions needed to manage its entities [23]:

- *match* to verify the conformance between a declared method and of call to a method (name, type - constructor/destructor, attribute/function);
- *are_valid_parameters* to test the compatibility between the actual and formal parameters at run-time (their number and their type conformance). This is an action which provides a sub-service for *match* action;
- *lookup* to find out the right method call taking into account the relationships between descriptions;
- *verify_overloading* to check that the overloading rules are respected;
- *verify_cardinality* to verify if the cardinality of the target relationship conforms to the value of parameter *Cardinality*;
- *is_same_type* to compare in the given context the exact equality between the two target types;
- *create_instance* to authorize the creation of an instance of a description.

Photo camera model may need the following actions to ensure the consistency of its meta-information:

- *block_film_open* to block the opening of film spot in order to avoid light exposure by mistake. This action is attached to traditional camera semantics;
- *check_battery_type* to match the correct battery type to a camera. The action is attached to the concept at the top hierarchy of cameras and it queries the *batteryType* characteristic;
- other similar actions may be: *check_communication_ports* or *check_memory_type*;
- *verify_memory_speed* to ensure that the memory stick attached to a digital camera supports the read/write speed of camera hardware;
- *verify_resolution* to check the capability between the resolution supported by the optical lenses and digital zoom of the camera and the firmware installed;
- *check_auto_focus_compatibility* to verify if the auto focus system is compatible with the optical sensors (it can connect and communicate to control and automatically adjust the focus).

Thanks to AOP (Aspect Oriented Programming) paradigm [66], it is also possible to insert new concerns (they are called aspects), with respect to the semantics of business-model. This is completely independent from the category of visit-entities from the potential applications (see section 3.4.4). Aspects were implemented in order to easily add new pieces of behaviour which are orthogonal to the semantics. Therefore, actions integrate aspects within the business model semantics (which are independent from potential applications).

The previous paragraphs showed that in SmartModels the description of the model semantics relies on actions which are described in a concept (at the meta-level) and on invariants for concepts or atoms and pre/post conditions for actions.

Aspects may be integrated within both assertions and actions. They are very similar to those which are dedicated to the visit entities (see section 3.4.4). Why may we need to equip the semantics of business models with aspects? A first answer is that the semantics of a business model may be complex enough to feel the need to separate its different concerns; this favours readability, maintenance and reuse. We do not propose a double cross-cutting of the semantics as for applications because it seems more useful to favour the use of assertions which is more relevant for the description of business model entities and for their interactions. Moreover, the number of statements which are necessary for the description of the semantics is clearly smaller than the number of statements dedicated to the description of facets (a complex application requires detailed customization of the facets because they address mainly the behaviour of the entities).

At a first glance, it seems more interesting to add dynamic aspects to applications, whereas aspects associated with business models are more static. However when a model is obtained by transformation of another one it may be valuable to have the ability to adapt the semantics encapsulated within one action (proposed in the original model) in order to make it compatible with the new model (this is why we provide aspects within assertions). For example, an aspect which implements an *around* assertion on an action and which is executed before the invocation of the action, may decide that according to the context of execution, the assertion in the new model is still relevant or on the contrary, it is not relevant anymore (this suggests that an *around* type of assertion may not be executed).

### 3.4.3. The reification-level

Here is the line of demarcation between the semantics (*the meta-level*) and the data of a business-model (*the reification-level*). As we anticipated in the previous section, an atom is the reification of entities of a business-model. Identifying the atoms of a domain is an essential task of a programmer.

The description of the business model entities relies on well-known concepts that may be found in most programming languages or meta-models. In SmartModels meta-model, the definition of an atom, the structure which supports the description of an entity, it is very close to the MOF "*class*" notion (the concept of class is, from our point of view, too much related to programming languages whereas business models require a more abstract concept). Then the features provided by MOF to describe the contents of a class (such as attributes, operations, generalization relationships) are sufficient to define most of the reification of an entity.

The designer of a business model may create atoms either for improving the structuring and factorization of information within the model hierarchy, or for describing atoms which have instances within applications. SmartModels provides a way to address those two issues; MOF does it through the notion of abstract class. If it means that the class must have at least an abstract method or that all the methods must be abstract, then we believe that this mechanism is not sufficient. In particular, some applications may be interested by some atoms whereas others are not; it is not the same thing to say that whatever is the context of use, one atom may not have instances because it is only partially defined.

A key principle in SmartModels approach sustains the belief that a more accurate information according to the atom status (see next paragraphs on genericity of atoms) will improve the readability of the code produced by generators, and the facilities that may be provided or not to the programmer of application according to it. The interest to be able to associate different status with an atom is even greater if the business model may import atoms from another business model.

Although not all atoms use this facility, each atom has its meta-information in the corresponding concept. An atom is seen as an instance of its concept (see

Figure 18). However, there are two axes of co-ordinates that distinguish between atoms:

- atoms which are generic or not. The support of generic entities (generic atoms) is an important issue for business models. The genericity is a reflection of the semantic-level which specifies if the meta-information of a given entity has or not parameters and characteristics (see section 3.4.2). A business-model designer may define entities which need semantic information (which becomes part of an atom definition) and they are called *generic atoms*. There are atoms which do not need additional customization besides their reification (their behaviour does not depend on parameters) and they are called *atoms without parameters*; and
- atoms which have instances within applications or not. The generic atoms may or may not have instances at the application-level. Now MOF [84] makes this distinction through the notion of abstract class. According to the arguments from the previous paragraph, that is why SmartModels has the notion of derived atom (see Figure 18) which is an instance of a generic atom obtained through relevant combination of values associated with the sets of characteristics and parameters which participate to the definition of its generic atom.

To exemplify lets turn again to the case of an object-oriented language. Let us take an example of one business model which is dedicated to record both the structures and semantics of Java programs. Possible applications with respect to this model may implement functionalities of programming environments (metrics, various wizards or editors, etc.). Possible atoms of this model represent, for example, *attribute, method, method parameters, modifiers*, etc... But the most interesting ones deals with the different kinds of classifiers and relationships (*aggregation*-like or *inheritance*-like). Most semantics may be encapsulated within classifiers and relationships and other atoms mentioned above may have a very minimal semantics mostly represented by their reification. This is possible because they are driven by the semantics associated with classifiers and relationships. In fact, there are several kinds of classifiers (e.g. *class, abstract class, inner class, interface*, etc.) and relationships (e.g. between interfaces inheritance - *extends*, between classes inheritance - *extends*, between interfaces and one class - *implementation*, between an abstract and a concrete class - *concretization*) in this business model. Then it is meaningful to be able to record their definitions as generic atoms (one generic entity for modifiers, one for inheritance-like relationships and one for aggregation-like relationships). All these properties are recorded in their meta-level through parameters (see section 3.4.2).

Therefore, the genericity comes from a set of hyper-generic parameters and a set of characteristics which records the differences and the commonalities between all the foreseen derived entities (this is the term which is quite often used in the state of the art to refer instances of generic entities, e.g. all the Java classifiers). Intuitively, *generic atoms* are quite similar to the concept of generic class in the Eiffel language and *derived atoms* are obtained through the relevant combination of values associated with the sets of characteristics and parameters which participate to the definition of the generic atom.

The photo camera example (Figure 17) clearly illustrates the difference between SmartModels atom types:

- The basic atoms (non-generic) are instances of basic concepts and therefore they are not customized in this model (this is the only reason why these atoms are basic here - it is not in the interest of our model and it follows the goal of keeping the example simple). However, they may be targeted by certain actions which have to control their implications in other generic atoms: *BatteryType* and *CommunicationPort*;

- The generic atoms are controlled by their semantics stored in the corresponding generic concepts: *PhotoCamera, DigitalCamera* or *MemoryType*. Some of their hyper-generic parameters have values assigned, but they cannot have instances in an application because they are still abstract - not all their semantics (parameters and characteristics) are specified. For example:
    - All photo camera from the reification model will support *opticalZoom* up to *10.0x* value and *Lithium-Ion* type of batteries;
    - All digital cameras will incorporate the following communication ports: *USB, Bluetooth and AV-out*;
- The derived atoms conclude the instantiation of their semantics and through polymorphism they can share common values in a common super-generic atom or even another super-derived atom:
    - *CyberShot* camera parameter values are: maximum *resolution* up to *8.1MPixels* and *digitalZoom* up to *2.0x* and memory characteristic is *MemoryStick*;
    - *PhotoSmart* type of camera is similar to the other digital camera derived atom;
    - *SDCard capacity* parameter specifies that this class of memories support up to *2GB* and their *speed* for data transfer is maximum *6MBs*.

At this level, the *actions* become more interesting because their roles are to check the atoms behaviour based on their semantic customization and their values at reification-level. For example, the action *verify_memory_speed* checks if the installed memory type in the *CyberShot* not only is compatible to this camera, but also supports the transfer rates of the camera's firmware.

Now, a new and clearer perspective can be obtained if the Eclipse EMF diagram of SmartModels (Figure 19) is analyzed in this context. The previous two sections highlighted the two main entities of SmartModels extension: the *concepts* and the *atoms*. The atoms represent the different types of elements which the approach can handle in the models - instances of SmartModels. The concepts are the meta-entities of these atoms. In this approach, the concepts are described through the generic parameters and the atoms are their instances which have a value for each parameter of the corresponding concepts. The derived atoms are atoms which are not provided by the meta-model itself by default. Therefore, they are specified in a model or a set of related models. In accordance with their definition, the atoms are abstract in general and derived atoms are concrete.

The generic concepts are composed of parameters (GParameterDefinition), but also of characteristics (GCharacteristicDefinition). A parameter represents a simple value (an Integer, Boolean or a predefined Enumeration). On the other hand, a characteristic is a multiple value, often a list of complex objects. The diagram does not include the hierarchy of the entities' basic types so it is not unnecessary crowded (i.e., parameter of Integer type or Tuple type of Integers). The same is true for the corresponding values hierarchy (GParameterValue and GCharacteristicValue). Therefore, the standard primitive types and values are replaced by EClassifier which means that the type is considered to be either a class (including Collections, or Tuples, ...) or a predefined type (Integer, Enumeration, ...). The complete hierarchy of types is mostly designed to be compatible with the hierarchy of the types of OCL. The characteristics make possible the representation of atoms with a more complex structure. Their value domain is normally bound to one or a set of parameters through their conditions or constraints. The final important elements of the concepts are the actions. They represent the executable part (algorithms) of the reification. They are almost always dependent on parameters or characteristics.

Figure 19 shows that an atom can specialize one or more other atoms just as a concept can also inherit from several concepts. A derived atom can be built from another atom or derived-atom. Here the interest is to be able to reuse a part of the values of parameters and characteristics associated to a derived atom, and therefore to have to redefine only the values which are different. This redefinition is controlled by the rules for redefinition associated with each parameter or characteristic.

The previous paragraphs showed that a concept represents the meta part of an atom: it establishes the common behaviour of a set of instances of the same atom, based on the value of its parameters and characteristics. The behaviour also depends on the structure of the atom (other features or methods). A concept has access to all the information associated with the structure of an atom (or the same information can be associated to several atoms). If there is a set of atoms that a concept generates, it usually limits the access only to their common part of the contents. As a result, it is interesting to use multiple inheritance between atoms to specify the common part of several atoms and, in specific situations, it is convenient to mirror the hierarchy of concepts on the atoms hierarchy.

If an atom is not explicitly attached to a concept by the meta-programmer of the model, it will automatically be attached to the same concept as its parent. This is also an opportunity to favour the reuse of the semantics and to preserve the consistency of the model.

### 3.4.4. The instance-level

In [106] and section 3.4.3, I explained the choice to use generic atoms instead of inheritance relationships for modelling the atoms and there are other interesting issues concerning them. Applying appropriately the SmartModels principles described in the previous sections should lead to a much more effective application building process with SmartFactory. The next paragraphs address the description of applications (this is also called application-level in SmartModels) which capitalize the atoms of the business model. As it has been mentioned earlier, we can distinguish two kinds of applications:

- Applications which describe model transformations, i.e.:
    - applications which may offer relevant information for refactoring or extending the model (creating new object-oriented programming languages or families of digital cameras or memory cards);
    - applications which may apply redefinitions (creating an extension of Java language to support multiple-inheritance, SmartModels EMF extension to support its entities or a new family of digital cameras which extend the maximum optical zoom to a higher value);
    - applications which may search for compatibilities between entities (like between cameras and memory types, ports, software ...)
- Applications which query, compute and update the instances of the business model, i.e.:
    - end-user applications like Eclipse EMF Plug-in or a software product line to produce photo-cameras or memories.

At this point, it is straightforward that the specification of those applications will slightly differ from classical object-oriented applications, even if both rely on the object oriented paradigm.

Intuitively, building an application is a process which consists of a set of traversals of the graph of atoms corresponding to a business-model. During this traversal, the behaviour contained in application facets is performed sequentially. While these facets are processed, it is possible to trigger the execution of aspects which allows integrating orthogonal services. The reification of both the business model and the application is handled by the meta-object protocol which contains also additional functionalities.

A type of traversal is the main entity which influence the way an application must be developed and it is called *facet*. The organization by facets of an application draws from SOP [48] and AOP [66].

A *facet* represents one concern of the application with respect to the business-model. This is a vertical cross-cutting of the applications (it is itself defined as an independent business model, so that it may also be associated with a DSL) whereas inheritance relationship would provide horizontal cross-cutting which introduces several levels of abstraction into the business model or the application (the model supports hierarchies of atoms, concepts, visit entities, facets and more generally of any first-class entity).

Each facet corresponds to a part of the treatment to be processed on one entity. Typically one facet of a given application would rather address the same set of atoms as the other facets (even if there is no constraint).

Therefore, SmartModels provides a new perspective over the application developing process. Because a facet represents a way of traversal through the model this approach is based on the Visitor pattern [87]. It is a design pattern which decouples the design of complex linked object structures from the design of functions that traverse these structures and that is why it fits very well in the approach (in the aspect oriented language AspectJ, a joint point is a particular location in the program where an aspect may be integrated).

In this way a facet is mainly compound by a set of visit-entities. A visit-entity implements a treatment on an entity of the business-model. It is described by an execute method which contains not only the behaviour of the visit, but also a mechanism for the description and the verification of preconditions and post-conditions. It has a mechanism for binding the business-model and has direct access to the properties of the entity it handles and to the context of the execution.

Also, a facet specifies:

- the business model that it addresses;
- how the business model is visited (the type of the traversal - see on page 75);
- the entities that are relevant according to the objectives, and possibly
- some additional technologies.

Technologies are defined independently from SmartModels (for example, by an API or a library of classes). They contain functionalities which allow defining more easily the application. For example, the DOM API is welcomed to manipulate XML representation of the business models. Also, the Eclipse EMF code generator [36] may be delegated to generate applications on Java platform. Other information may be added in order to generate visit entities which fit exactly to the expectations of the programmer. Indeed the source code generation is essential to our approach because it allows the programmer, to focus only on the visit entities that are addressed by the facet but also to be assisted for the description of their behaviour.

Organizing facets by visit-entities offers a couple of very important advantages:

- It is a very flexible framework which allows an easy transformation of the behaviour of the applications;
- It is dependent on the structure of the business-model and it can traverse it at different levels of granularity;
- It has the possibility to carry information during the traversal either up (toward the root of the facet - by default the starting atom is the root atom of the business-model) or down (toward the dispatched visit-entities);
- It has the capability of inserting orthogonal concerns and of testing invariants, preconditions and post-conditions.

The facets are classified according to the type of traversal. Thus, there are three types of ways to run through the graph of entities, all of them deep-first traversal, but the approach is very open and flexible for more kinds of traversals to be proposed depending on the needs of the programmers:

- *plain* facet: it consist of a set of visit-entities each one of them corresponding to one and only one entity of the model. The possibility to explore the values of all the visible properties of the entity and to control the navigation policy is an open capability shown to the programmer. In this way, he can decide what to query and where to redirect the traversal.
- *detailed* facet: it is similar to the previous type, but the granularity is enhanced as a visit-entity corresponds to a property of an entity. In this way the detail facet will contain as many visit-entities as many properties the target entities have.
- *hide* facet: the main difference compared to the other types is that this facet hides the traversal and thus the programmer cannot control the traversal. On the other hand there are very important advantages: the use of hide facet is less complex and it favours the reuse of the application semantics of atoms which evolve. It also can be plain or detailed.

Invariants, pre-conditions and post-conditions are assertions seen as first-class entities and like in programming languages such as Eiffel, they can be evaluated at the beginning and at the end of a method; they determine whether the visit of one atom succeeded or failed. A future project is to investigate how to describe them using the OCL [82].

Section 3.4.3 foreseen the relevance of aspects at the application level. An aspect can be attached to several visit-entities or to several facets (it will be applied to all their visit-entities) of a given application. The association between aspects and the set of visit-entities is absolutely free. They will be performed at some points of the execution of one or several visit-entities, one or several facets (for example, to check the validity of a constraint, to load data, to check access rights or to trace a method call).

The expressiveness related to the specification of this set depends only on the language dedicated to the definition of joint points within the facets. This language does not require the expressiveness proposed for example in AspectJ [65] and does not address program structures, but atoms. For example, in a business model, if only one visit entity is associated with a given atom, then there will be only one joint point per atom. But in the same way as it is proposed for AspectJ, it is possible to customize one visit entity in order to integrate an aspect at different moments: before the invocation of the visit or when its execution starts or ends. It is also possible to distinguish (already implemented in the prototype) two kinds of execution ends: the execution fails (one assertion is not satisfied) or the execution succeeds (all assertions are satisfied). The expressiveness of the handling of aspects and in particular the description of joint points should be handled by an independent business model which like facets, could be associated with a DSL; it is another example of the self extensible capability of SmartModels.

### 3.4.5. Methodology to Describe a Model

Section 3.4 explained every entity of the approach with respect to the level it manifests. Based on this presentation this section resumes the SmartModels methodology to describe a business model. This is a five-step process:

- To identify and to specify the basic atoms of the model,
- To identify the generic atoms,
- To define the criteria of genericity (the hypergeneric parameters and characteristics),
- To specify the actions attached to generic and non-generic atoms,
- To specify the instances of the generic atoms (derived atoms).

The three last steps deal with the specification of the meta-level (the concepts). Typically, these steps must be performed by an expert of the domain, because it represents a part of the knowledge of the business model.

As a result, these may be reducing to three main entities that a modeller has to define in order to build a SmartModels model:

- to elaborate the list of *Atoms*;
- for those who are generic to add their semantic information (the hyper-generic parameters, characteristics and actions) in the list of *Concepts*;
- to compose the list of the *DerivedAtoms* setting their semantic values.

## 3.5. A Set of Empiric Rules for Model Oriented Programming

As it has been mentioned in the previous sections, model oriented programming introduces a new level of abstraction (the model) which acts as an autonomous entity that may receive queries from satellite applications. The specification of both the model and the applications may use, for example, object-oriented and/or aspect-oriented approaches.

Each application is built around at least one business model. SmartModels approach addresses the most frequent case where one business model is predominant and on which different concerns of an application are plugged-in. This is very similar to approaches by separation of concerns [59] [66].

However, in model-oriented programming, concerns are attached to a business model instead of being weaved into object-oriented applications which may be executed with or without these concerns. The model has its behaviour (its semantics), but it does not invoke itself any treatment. On the contrary, the semantics of the model are addressed only when applications query the model entities in order to match their requirements.

In the context of Domain Driven Development (DDD) [27], a business model may support two main categories of applications:

- those dedicated to the computation and/or the update of information recorded by the instances of models; their methodology is close to information systems, and

- those which deals with the transformation of the model and which are particularly relevant in the context of MDA.

Model-oriented programming is fundamentally different from other paradigms such as object-oriented programming (OOP). It breaks the supremacy of programming languages: the model is now the key point whereas the formalism used to describe its instances plays a minor role. This is the consequence of the collaboration between MDA [86] and generative programming [26]. Altogether, these two paradigms contribute to link the model and its formalism(s), and this favours the emergence of Domain Specific Languages (DSL) to make models executable.

A new approach for the development of software must ensure that software engineering skills are covered and improved in comparison with object-oriented and aspect-oriented approaches. Reuse, evolution capabilities and robustness of both models and applications must be addressed very carefully by model-oriented programming.

In this context, at the end of this chapter which presented SmartModels, a model-oriented approach, this section uses the opportunity to raise for discussion a set of empiric rules. They are an attempt to put in a concrete form, lessons we found important in order to build model-oriented software. It is a synthesis of principles and advices that were born in the midst of this thesis research.

Therefore, this section is a proposition of nine rules classified in two categories according to their purposes and applications: conceptual design and implementation. For each one of them we also point out how they address important and practical issues for programming model-oriented.

### 3.5.1. Rules for the Design Approach

***1: Business Model as a first-class entity of the development process.***

A business model relies on a data model and on a semantic model. The data model contains the description of the entities involved in the business model whereas the semantic model describes the interactions and the constraints between those entities, but also their behaviour with respect to the business model know-how.

A model is considered by applications as a whole and it is queried for its contents; it constitutes a new level of abstraction which favours global operations such as transformation or introspection. Both of them query the model entities in order to reuse its business know-how or to evolve both model and programs.

***2: A triple independence between the model, the application and the technology.***

A model is not an application. The model encapsulates the description of its behaviour (its semantics), which must be independent from any further use. This property will ensure that a business model is reusable independently from the applications that may address it.

Moreover, an application or a business model must be designed independently from the software platform on which the application will be executed. It is only at the very last moment that the binding with the platform technology must be made. This property allows the business logic to be used whatever technology will appear in the future.

***3: Support of generic entities***.

Typically models may address lines of products and more generally a set of entities that may have commonalities and differences, but which have close semantics; they must be designed as generic entities which may be easily derived.

A quite common situation is that business models address a few key-entities which are defined according to a large number of basic entities; very often, the key-entities correspond to generic entities. Then it is particularly important: that generic entities provide a clear vision of their semantics because they deal with a significant part of the model semantics. Object-Oriented languages, like Eiffel, proved that the support of generic entities (equivalent to the generic model paradigm) is an interesting approach to ensure reuse and evolution.

***4: Clear separation between semantic and data models***.

The domain-specific know-how is encapsulated in business models through the data model (reification and structuring by the entities) and the semantic model (behaviour of those entities).

To be able to reuse the semantics when the data model evolves is an important issue. This is particularly important in the context of model transformations where semantics must evolve accordingly to the data-model (in the most automatic way). Model-oriented programming must provide a clear separation between the description of the data-model and the description of its semantics.

***5: Orthogonal handling of concerns***.

Rule no. 2 infers the importance of separation of concerns between the model and the applications. The model is under the responsibility of an expert which captures the domain-specific know-how, whereas the applications are handled by the programmers. But separation of concerns must exist also within the model and within the applications themselves.

According to the business model, the needs are twofold:

- the semantics may be complex enough and require some modularization, and
- the pieces of semantics which are orthogonal to the original semantics must be straightforwardly carried out.

At the applications level the requirements are even more important. An application may contain different subjects which have to be smoothly composed for building it up. Moreover, an application should be able to take care about the evolution of the environment (which cannot be foreseen in advance), without changing the application core.

### 3.5.2. Rules for the Implementation Approach

***6: An adequate balance between declarative and imperative programming***.

Semantics of models should be described as much as possible in a declarative way in order to specify what is expected (the "*what*") but not how it is made (the "*how*"). This is one of the most important issues addressed by the MDA approach [86]. But, it is not acceptable to carry this approach to the breaking point where the description relies on very complex formalisms, difficult to read and to understand. A compromise is necessary between the "*all declarative*" and the "*all programming*".

***7: Support of domain-specific languages***.

A clear distinction has to be made between the expressiveness of a business model and the language (textual, graphical, etc.) used by the designer for the specification of the different pieces of this model. Moreover, model-oriented programming tends to come closer and closer to the general public (ubiquitous programming), so that the need to provide "*languages*" dedicated to one business model and even to one application becomes more and more important. Generative programming and MDA provide a good support to achieve this issue.

***8: Openness of the development process***.

To provide a meta-model and a set of related mechanisms that answer to any need of any kind of business model is utopian from our point of view. We promote the idea of an unified approach with very few built-in mechanisms, but which can be easily adapted to further needs of modern applications.

In particular, it is important to be able to customize the way to query information according to the context of use. In other words, the generation and handling of an executable business model must be customizable. In our approach, all the key-concepts which participate to the description of both the application and the business model in order to make it executable are first-class entities.

***9: Self-extensible capability of the approach***.

Model-oriented programming requires a meta-model which captures the description of both business models and applications, as it is mentioned in the previous rules. This meta-model may be considered as a particular model. As it is explained in rule no. 7, the specification of the different parts of this meta-model may rely, for example, on a dedicated language. But many other needs required for the development of applications may appear.

In particular, modern applications should be available as components that may interact with each other. It is important to make the approach self-extensible. In other words, it must be able to include other applications and models built thanks to model-oriented programming (that means built with the approach itself). For example, to handle components, a correct approach would be to design a model.

With these nine rules, we attempted to set a framework for model-oriented programming. We promote the idea that an approach which intends to implement model-oriented programming should try as much as possible to match the requirements proposed by these rules. This chapter proposes our approach, SmartModels, and next three chapters demonstrate how it addresses these rules.

## 3.6. Conclusion

This chapter is the central point of this thesis because it presented the SmartModels approach which is dedicated to contribute to strategies to increase software quality and productivity: the key elements and the methodology to describe a model. It tries to be part of the new trends in current research communities for software engineering. It proposes another way of developing software: *Model Oriented Programming* (MOP). It is based on the Domain Driven Development track (DDD) [27], which relies on several paradigms such as object-oriented technology, languages for components, MDE, approaches for the separation of concerns, and generative programming [26]. It intends to enrich these approaches in order to make easier the development of domain specific applications.

The main SmartModels objectives are on the one hand, to clearly identify, thanks to a meta-level, the semantics of concepts used for the modelling of a given domain, and on the other hand, thanks to approaches of separation of concerns and generative programming, to provide, in a modular way, the applications related to this domain.

Figure 18 shows the three levels of SmartModels meta-model architecture and section 3.4 introduced every entity with respect to the level it manifests. The presentation is decorated with two examples (a meta-model for object-oriented languages and one for the domain of photo-cameras) which illustrate the flexibility and expressiveness of the approach.

The meta-model of SmartModels is a core which acts like a foundation for various applications. Chapter 4 takes the discussion one step further and proves that this approach, which is based on customizable genericity, provides the needed expressivity for modelling product lines. Therefore, one of the most important applications, and also one of the main purposes of the approach, is to address the description of a line of products and this chapter gives full details on how to use SmartModels features in order to contribute for a better modelling of SPL.

# 4. Application to Software Product Lines

The Software Product Lines (SPL) approach is a transposition of the technological lines of production into the world of the software. The SPL goal is to minimize the costs of software implementation for a particular domain and the idea is simply this: instead of developing each application of the given domain separately, this approach proposes to construct all the related applications from reusable similar components.

The heart of the product line approach is based on a methodology for strategic reuse of requirement specifications, software architectures, design models, source code, systematic product line creation and improvement, components, test cases, and the processes for building families of systems and modelling their artifacts (see section 2.4).

The first difficulty related to this approach lies in the need to design an architecture which should make it possible to define several products. The products of a line are characterized by their similitude (also called common points), but also by their differences (also called *variation points*). The management of this variability is one of the key paradigms of the product lines approach.

For example, in a line of vehicle production, cars are manufactured starting from a set of common elements (wheels, steering wheel, windows, headlights, etc...), but each model can have some characteristics which differentiate it (the horse power of the engine, fuel type, gearbox type, climate control ...). In the world of software, the differences can appear in the same way, according to technical requirements (the choice to use a particular encryption algorithm), commercial (the marketing or client service rules), etc.

Another difficulty lies in the use of a product line. To some extent, the construction of a software product (this is usually addressed as *product derivation*) consists in making a set of choices (*derivation*) vis-à-vis to the variation points defined in the product line. Evidently, some choices are incompatible between them. In the above car example, it is clear that a car generally has only one engine. Then it is necessary to choose between a petrol or diesel motorization. In the same way, a particular choice at the time of the software derivation can exclude some alternatives. For example, the choice of a convertible car with folding roof will exclude the possibility of choosing a retracting roof. Therefore, a product line must also integrate constraints which enforce coherence in the product line and which make it possible to facilitate the choices at the time of derivation.

This chapter proposes a technique of modelling the product lines based on SmartModels, presenting the way this approach integrates static and dynamic concerns on the models and how to insert constraints for coherence (section 4.1). Section 2.3 already introduced how UML responds and what entities it offers to the modelling of product lines. Section 4.2 brings this work to a close and identifies some perspectives.

## 4.1. Modelling a Framework for Generating On-Line Assessment Software Solutions

In order to illustrate SmartModels approach for building software product lines, this section will introduce the example of how to model a framework for generating on-line assessment software solutions; particularly, it targets the process of developing e-learning deployment tools. This is inspired from the domain of web-oriented course management systems to help tutors to customize effective online learning communities and their evaluation.

Globalization has considerably affected the way education can be provided. Every knowledge provider institution has to publish and offer its expertise for a wider readership in order to stay on top. One of the prime-time opportunities is to build web-based software solutions in order to support distance learning. There are so many ways to organize your class and professors can imagine so many ways to evaluate students that the complexity of such a software system can be hard to model and to implement. It is also very hard to foresee future education forms.

One of the main problems the companies face today is that even if the analysis and design stages provide a perfect model, the programmers have to make a lot of compromises when trying to implement the model using a specific programming language and mapping to a specific platform. This problem applies to the process of developing e-learning deployment tools when trying to encapsulate all type of possible knowledge presentations or questions from an assessment.

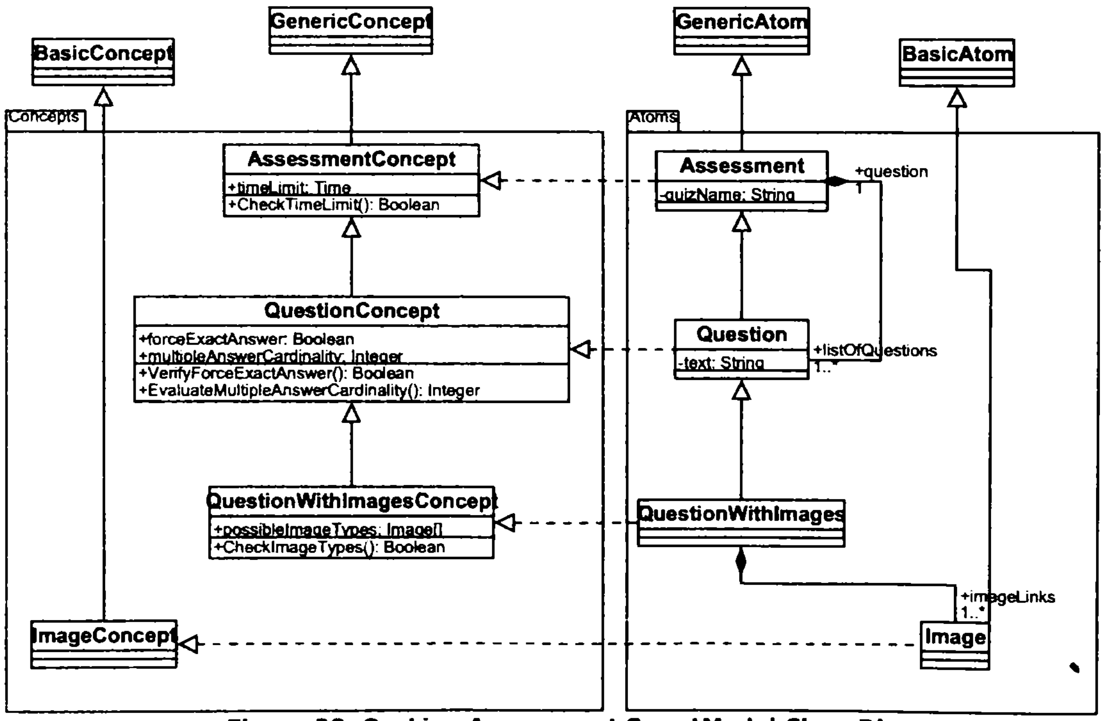

BasicConcept
GenericConcept
GenericAtom
BasicAtom
Concepts
Atoms
AssessmentConcept
+timeLimit: Time
+CheckTimeLimit(): Boolean
Assessment
-quizName: String
+question
1
QuestionConcept
+forceExactAnswer: Boolean
+multipleAnswerCardinality: Integer
+VerifyForceExactAnswer(): Boolean
+EvaluateMultipleAnswerCardinality(): Integer
Question
-text: String
+listOfQuestions
1..*
QuestionWithImagesConcept
+possibleImageTypes: Image[]
+CheckImageTypes(): Boolean
QuestionWithImages
+imageLinks
1..*
ImageConcept
Image

**Figure 20. On-Line Assessment SmartModel Class Diagram**

SmartModels tries to reduce this gap between the design and implementation and to ensure the independence between the model, future family of applications and the technology (which evolves so rapidly). Through its small kernel and a set of basic entities (see chapter 3), it provides a framework to describe models and chapter 5 presents SmartFactory, a software factory to automatically generate code as much as possible.

This means that the applications may be re-generated at any time if the model or the technology evolves and also the model instances can drive the behaviour of the application at code generation time or at run-time (the regeneration of the code even in the situation when there is a need for integration - code was manually added in the previous version of the application - relies on the code generator; see section 5.3.4). Therefore, SmartModels applies MDA for SPL and SmartFactory is reusing the know-how of the most widely used platforms for building integrated development environments (IDEs) like SmartTools [7] or Eclipse [35].

Following the SmartModels methodology to describe a model, Figure 20 presents the SmartModels class diagram for building an automatic creator and generator of an on-line assessment software. The designer may identify the next main entities: the assessment, the different types of questions and other necessary basic elements, like: images and answers. Next sections will explore SmartModels techniques to model a product line for this software through the above example.

### 4.1.1. Handling Meta-Information for Complex Family of Entities

One of the hardest parts of creating a flexible tool for developing on-line assessment is that there are many ways a professor can imagine the evaluation of students. One can decide to create multiple-choice questions and require correct answers on all choices to mark all points. Another professor can decide to mark just the good answers. Others may even think of a weight for each answer and subtract points if a student makes wrong choices.

Let's imagine the requirement to automate the evaluation of a quiz which is expected to be filled with exact answers. This can mean checking for spelling mistakes in case of a free-form text question or labelling correctly a set of images.

Should the structural entities of our model which describe the different types of questions of the quiz be equipped with all the information about all possible ways to evaluate them? SmartModels is a framework which makes a clear differentiation between the semantic information and reification of the families of entities of a domain. In this example, this approach can unambiguously separate the description of the structural features of each question type from the concerns that deal with the process of evaluating them.

This means that the user will define atoms only to encapsulate the different structural features of each question type without having to concentrate on the way they will be evaluated or mixing each type of question with all its possible evaluation manners. On a meta-level, the user can concentrate on creating rules to evaluate a quiz. More than that, he can create rules which apply at the level of a single question or a set of question types (i.e., how are multiple-choice questions marked) or even rules for the whole quiz (i.e., setting a time limit to complete the assessment or one particular question).

### 4.1.2. Variation through Polymorphism and Abstraction

Product variation through polymorphism and abstraction (through inheritance and composition in object oriented languages) are important issues in the SmartModels approach, but this section will not over-emphasize them because they are fundamental and in the meantime well-known mechanisms inherited from OOP approach. However, it is important to mention that modelling through SmartModels (which is built in the context of object-oriented technology) implies that a meta-programmer can make use of all the benefits of object-oriented fundamental concepts for specializing families of products.

The example of a framework to create on-line assessments includes the use of two relationships in order to organize the model entities:

- *specialization*: for example, *QuestionWithImages* is a kind of *Question* which contains one or more images on the subject of the problem raised by the corresponding question;
- *composition*: for instance, *Assessment* contains a set of *Question* entities and *QuestionWithImages* may refer to a set of images.

In this way the meta-programmer can continue to enrich the model by adding other heirs to describe new types of questions. But more than that, this statute of SmartModels applies both at the level of concepts and atoms. This is one of the main reasons for which this approach can be applied for creation of families of entities. As a consequence, here are some of the most important advantages:

- at the meta level - to extend or to refactor the model and to enhance the entities semantic information, and
- at the reification level - to enhance their structural properties and to reuse entities and their properties in order to describe more specialized entities in the same model or even in other closely related models (inheritance at the level of models).

As it was already mentioned, a quiz may contain questions dealing with text and/or images. This section considers only the case of questions with text and images (imagine a question where a student has to match a set of names or statements from one column with the corresponding images on the other column). Figure 20 shows that this model diagram needs just little change in the hierarchy to encapsulate meta-information of other different question concepts. *QuestionConcept* addresses the semantic information of a general text type question and its specialization, *QuestionWithImagesConcept,* concentrates on more specific characteristics dealing with image manipulation. In this way, the instances of this concept (the corresponding questions with images atoms) will use semantic information about the text column of the question through inheritance.

### 4.1.3. Variation through Parameterized Genericity

Section 3.4.2 presented SmartModels flexible management of genericity through concepts which encapsulate the meta-information of an entity. It also detailed the role of the hyper-generic parameters to describe the semantics and the actions which drive them to guarantee the semantics control over the behaviour of the derived entities.

For example, a model built in order to encapsulate the structures (entities) and semantics of a tool to create on-line assessment solutions (i.e., quizzes) may present parameters like:

- *MultipleAnswerCardinality* which tells if a question corresponds to a single (1) or multiple possible correct answers (*), or
- *ForceExactAnswer* which expresses the requirement to accept only precise answer (*true*) in case of expecting a name or checking for spelling mistakes, or if it is interpreted together with the first parameter it can have the meaning of accepting an answer only if all choices are correctly set, or
- *TimeLimit* which adds the aspect of time limitation for the specified assessment or question,

and characteristics like:

- *PossibleImageTypes* which indicates the list of accepted picture file types (it assumes that images are reified through basic atoms in this model).

In this context it is important to remember the SmartModels distinction between *basic and generic atoms* (see section 3.4.3). An atom is generic if its meta-information presents parameters and/or characteristics. If an atom does not need additional semantics besides its data-model, it is called *basic* and it will have direct instances within applications.

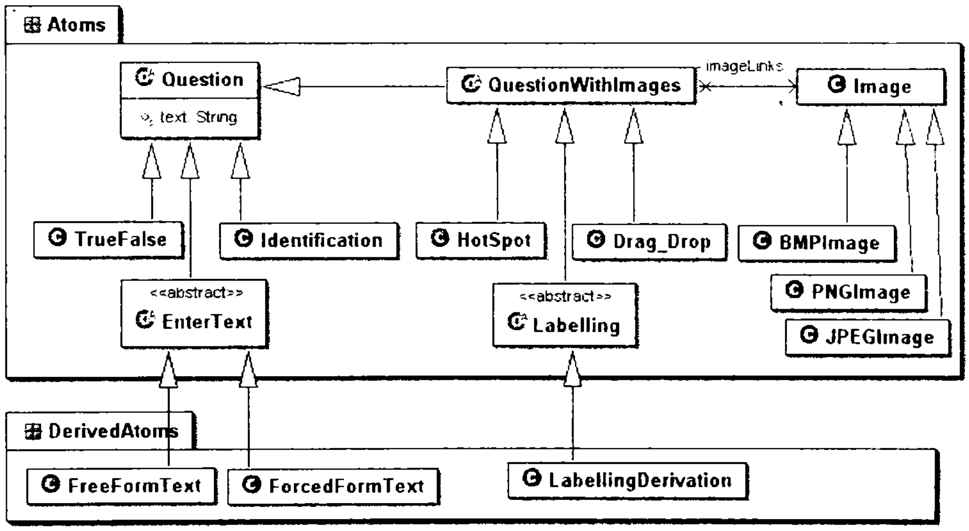


**Figure 21. On-Line Assessment Tool Atoms**

Therefore, the assessment tool solution identifies the following atoms (see Figure 21):

- *basic atoms*: image types, answers (this choice applies to this particular model – these atoms did not raised any interest to add more semantic information);
- *generic atoms*: a question with hyper-generic parameters, characteristics and actions like those presented in the previous paragraph. Thus, its heirs can imagine modelling all sorts of question types: hot-spot (allow student to answer by selecting an image from a set), forms to entry text, drag and drop images (set up an image which can be dragged over a list of other possible

corresponding images), labelling (label a set of images to match their text descriptions), text identification (for example, having the purpose to identify each spelling mistake in a passage), true/false questions. The list of generic atoms can be continued to be designed so the model can benefit of the advantages of polymorphism.

### 4.1.4. Variation through actions to manage dynamic aspects and constraints

What if after deploying the first version of the tool for developing on-line assessments, the requirements are updated and the designer needs to add a new mode of organizing the exam which was not planned at the model design stage, for example: to enforce a time-limit. Typically a professor should be able to stipulate this option when he creates the quiz, even if it was not included in the original model, because it should be checked at run-time. In a classical object-oriented approach, it would lead to considerable changes in the structure of entities and in their behaviour in order to implement this enhancement.

Thanks to the aspect-oriented approach integrated in SmartModels, it provides the opportunity to attach actions (for example: *CheckTimeLimit*) to each concept to dynamically control the behaviour of entities. This opportunity joined with the fact that we can benefit from inheritance, increases the level of flexibility of the model: a professor may think either to set a time limit on individual questions if he likes, or a global timer for the whole exam if the time-limit parameter is also specified at the level of the assessment (see Figure 20).

### 4.1.5. Variation through Atoms Derivation

A prerequisite of this section is to remember the SmartModels distinction between *generic and derived atoms* (see section 3.4.3). A derived atom is an instance of a generic atom obtained through relevant combination of values associated with the sets of characteristics and parameters which participate to the definition of its *generic* atom (see Figure 18). Derived atoms are another means provided by SmartModels to enrich the model and capture in the modelling phase as much as possible the commonalities and variabilities of the domain entities.

Considering again the assessment tool example, Figure 22 and Figure 23 present a couple of possible derived atoms reifying questions with different properties (text or labelling type questions):

- a free-form text edit derived atom is a text form question atom may have the following instances of its meta-information (see section 4.1.5 for a detailed diagram and presentation):
    - forced exact answer: *false*;
    - time limit: *none*.
- or a type of image labelling can be described by:
    - multiple-answer cardinality: 4 (the correct choices can be up to four);
    - forced exact answer: *true*;
    - time limit: "00:05:00" (no more than five minutes);
    - image types accepted: basic atoms like *JPEGImage*, *PNGImage*, *BMPImage*.

In this context, the designer can equip the on-line assessment deploying tool with more question types in two ways:

- either to create new atoms (creating new hierarchies of atoms in case of new entities from the target domain or creating heirs of existing atoms to obtain specialized atoms through inheritance) or
- to derive new atoms from generic atoms in order to create new entities through a relevant combination of parameter and characteristic values. The number of combination possibilities is therefore limited only by the richness of the semantic information described through parameters, their type ranges and relevance in relation with other parameters from the same conceptual tree.

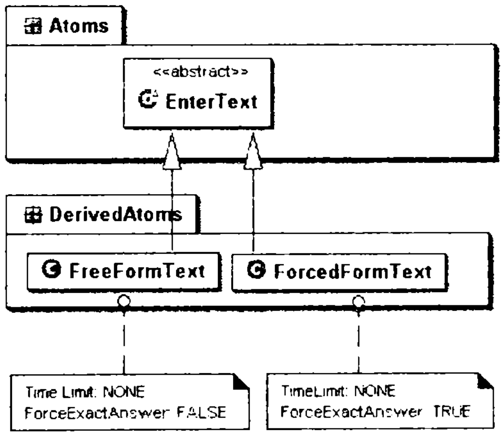


**Figure 22. Edit-text derived atoms**

Now the effects of setting the *ForceExactAnswer* parameter value (with true or false values, or we can even imagine setting down a default value) when creating a new instance of *ForceFormText/FreeFormText* derived-atom can be seen at run-time. Other variation of this edit-text type questions may have a time limitation which can also be specialized to be an exact period of time or a fixed date and time value (a professor may not want to set a timer on the exam editing, but to set date and time limit until the assignment may have to be submitted).

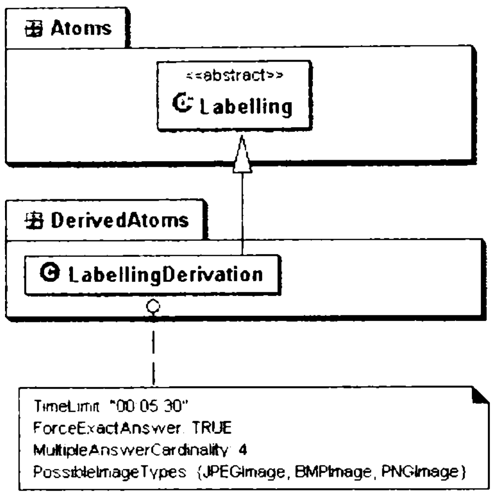


**Figure 23. Labelling derived atoms**

The other interesting illustration would be to consider a labelling type of question when the student has to associate a set of names or statements to a set of images. A derived labelling atom can be obtained through a combination of values of its concept parameters. If *MultipleAnswerCardinality* is set to "4" then this means that there will be a question where there will be four choices to be presented to the student. The use of *TimeLimit* parameter can set a timer on this question and *PossibleImageTypes* characteristic may enumerate the types of images that can be displayed by this tool (*{JPEGImage, BMPImage, PNGImage}*).

Enriching the semantics of the model example helps to imagine many other possible derivations. For example, an easy and flexible way to manage the creation of labelling questions, is obtained by adding a new parameter to specify the number of labels and the number of images to match. In this way a professor may create a question with ten labels from which to choose only the four valid names for the right column images.

Certainly, the choice of the concepts hierarchy as well as the associated parameters and characteristics, and the atoms hierarchy may be discussed by an expert of the domain.

## 4.2. Conclusion

SmartModels approach together with its prototype (see chapter 5) wants to form a possible and feasible way to apply MDA principles to build SPLs. This can be possible in Eclipse Environment [35] and EMF Project [36] and based on the example described in this chapter, this approach proposes a way to address meta-modelling issues, extending the know-how of the Eclipse platform. SmartModels is a MDA approach which provides a framework to create models that capture information about a business-domain independent from a technology, platform or programming language.

The main difficulties when building SPL is to deal with the variation and derivation of the products. Here are the major opportunities provided by SmartModels approach in order to handle these complex paradigms:

- a clear differentiation between the semantic information and reification of the families of entities of a domain;
- a flexible way to capture meta-information through concepts (parameterized genericity through parameters and characteristics) to handle the semantics for a family of a domain entities;
- making use of all the benefits of object-oriented fundamental concepts (polymorphism and abstraction);
- adding actions for the management of dynamic aspects and constraints which guarantee the semantics control over the behaviour of the derived entities;
- deriving atoms as another mean provided by SmartModels to enrich the model and capture in the modelling phase as much as possible the commonalities and variabilities of the domain entities.

Due to the growing interest in educational technologies, the aim of this chapter is to explore SmartModels' approach for building software product lines and to illustrate it through the example of modelling a framework for generating on-line assessment software solutions. This example proved the expressiveness of SmartModels approach to capture within a model as much meta-information as possible. Chapter 5 will then show the next step – how to automatically generate code on a specific platform and map to an up-to-date technology.

# 5. The SmartFactory Prototype

The purpose of this chapter is to present the implementation of the SmartModels approach, called SmartFactory. It is a tool support which was built in order to experiment and validate the approach. It deals with important implementation issues and represents an interpretation of SmartModels meta-modelling approach.

Two implementation prototypes are presented and I believe that both help to better understand the interest of SmartModels. The chapter starts with a short presentation of the first prototype developed using SmartTools platform (see section 5.1). Then, the second prototype, more elaborate, is implemented on the Eclipse platform (see sections 5.2 and 5.3).

This work stands between the model approach and the AOP (Aspect Oriented Programming) [66] dedicated to DSLs (Domain Specific Languages), but in a broader context, as it uses the concept of software factory. This later includes other notions for the design of applications such as software components for distributed applications.

## 5.1. First attempt to build the prototype

I will address all the main aspects, from the description of the meta-model to the achievement of the first operational system. Figure 24 describes the implementation of this prototype which is based on the SmartTools technology (see [7] [106] [107]). It represents the first step of the research conducted in the DDD [27] framework. To notify the flexibility and the rapidity of the development with SmartTools: the first prototype was achieved in three months. Thanks to it, I could prove the feasibility and the importance of SmartFactory, but I also obtained some very insightful and helpful feedbacks in order to improve it. The second prototype of SmartFactory is described in section 5.3.

The meta-model, whose key aspects are presented in section 3.4, is described with the *absynt* [7] language which is close to the BNF. From an *absynt* model instance, SmartTools automatically generates the reification of the business-model in Java. Both the AST (Abstract Syntax Tree) - of one business-model - and the instance of the generated AST - an instance of the business-model - are described in XML. In order to provide an easier way to input related information, it is possible to use the language *cosynt* which enables to define a concrete syntax (textual or graphical) with one or the other AST.

In order to preserve the flexibility of the prototype (maintenance, evolution, etc.), SmartTools adapted different visitors (fourteen in the first prototype) [107]. In fact, one visitor is created for each task to be performed; each of them could be replaced or removed according to the evolution of the prototype. It is not interesting to list all of them, but they are classified in three main generation tasks: the Java classes related to the business model,those related to applications and other to data

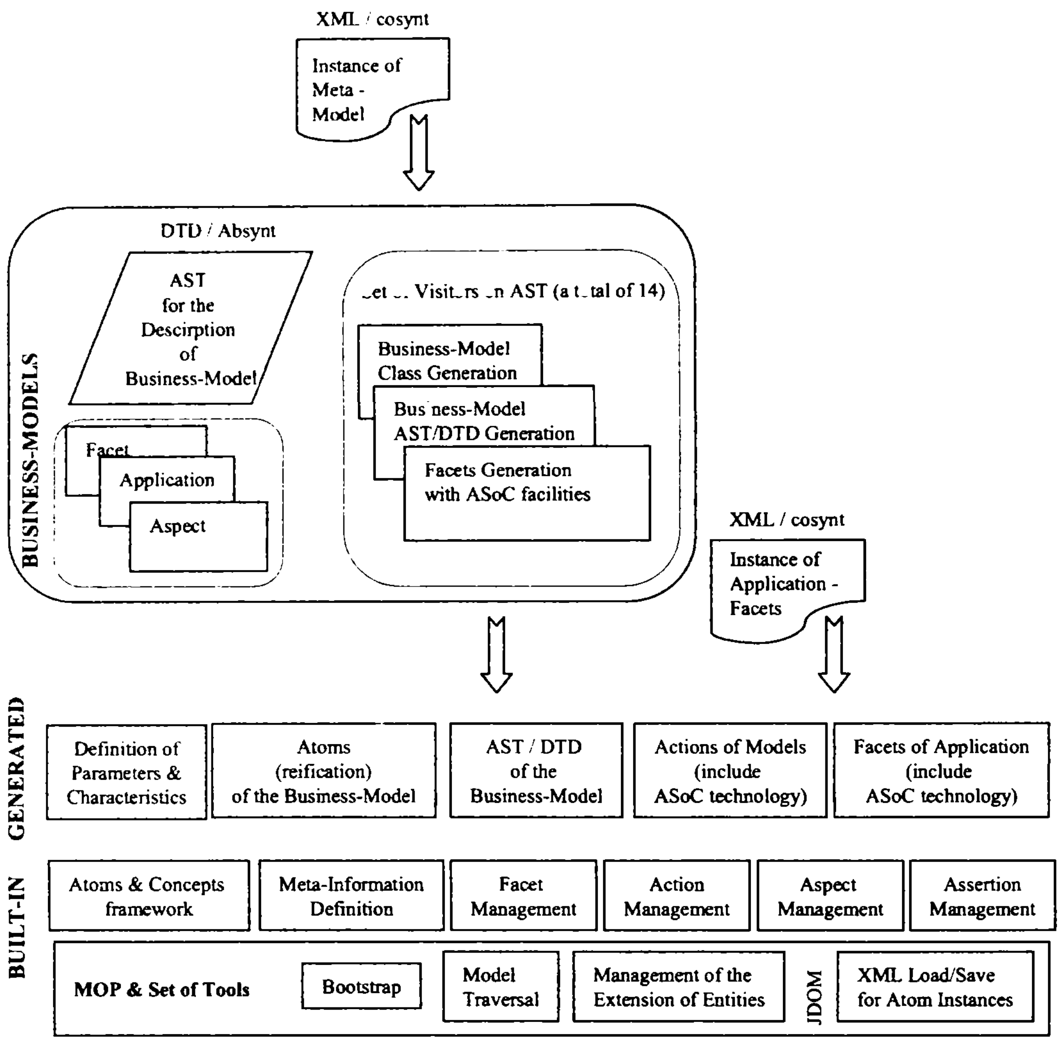

XML / cosynt
Instance of Meta - Model
DTD / Absynt
AST for the Descirption of Business-Model
Set of Visitors on AST (a total of 14)
Business-Model Class Generation
Business-Model AST/DTD Generation
Facets Generation with ASoC facilities
Facet
Application
Aspect
BUSINESS-MODELS
XML / cosynt
Instance of Application - Facets
GENERATED
Definition of Parameters & Characteristics
Atoms (reification) of the Business-Model
AST / DTD of the Business-Model
Actions of Models (include ASoC technology)
Facets of Application (include ASoC technology)
BUILT-IN
Atoms & Concepts framework
Meta-Information Definition
Facet Management
Action Management
Aspect Management
Assertion Management
MOP & Set of Tools
Bootstrap
Model Traversal
Management of the Extension of Entities
JDOM
XML Load/Save for Atom Instances

**Figure 24. Implementation of SmartFactory with SmartTools**

model representations (AST/DTD). For example, according to the Java classes related to the business model, there are seven instances of the Visitor design pattern: definition of hyper-generic parameters and characteristics, specification of actions, description of assertions, reification of both generic and non generic atoms, description of concepts.

The output of those generators is classes which rely on built-in sub-hierarchies and on one *mop* which includes facilities for:

- handling the set of instances of an atom (its extension), taking into account the polymorphism,
- loading/serializing instances of the model from/into XML files,
- browsing through the instances of business models.

Some of the sub-hierarchies encapsulate mechanisms for handling assertions, actions, aspects and facets. Others provide core hierarchies dedicated to the specification of business model meta-information and to the integration of both reification and meta-levels with the *mop* (see section 3.2.1).

The next two sections address the implementation of the second prototype. Section 5.2 describes the Eclipse platform and tools and section 5.3 presents the prototype. Section 5.2.1 particularly explains the motivation for a new prototype and lessons learnt while developing on SmartTools platform in contrast to Eclipse platform.

## 5.2. Eclipse Platform and Eclipse Tools Project

This section presents the Eclipse platform and projects used to build the new prototype of SmartFactory. It will draw a parallel between Eclipse and SmartTools platform, reasoning why I shifted to Eclipse and presenting the roles of its new tools. It will introduce the Eclipse Modelling Framework (EMF) modelling and generating tool framework with a short introduction about the generic template engine Java Emitter Templates (JET), the Graphical Editor Framework (GEF) tool, the development process of my Rich Client Application (RCP) on Eclipse platform.

### 5.2.1. Eclipse Platform

The SmartFactory first prototype was build in the framework of SmartTools. This was the first step of the research conducted in the Domain-Driven Development framework [27]. It is a development environment generator that provides a structure editor and semantic tools as main features. It was built on Java and XML technologies as a research project in the INRIA laboratory (`http://www-sop.inria.fr/smartools/`) from Sophia-Antipolis, France. Therefore it offers support for designing of new software development environments for programming languages as well as domain specific languages defined with XML.

In order to be able to define a new language, SmartTools uses its own abstract and independent defined formalism called AST (Abstract Syntax Tree). Because of this dependency on its own formalism, because SmartTools is not yet a very well-known project in the international research communities and because it does not have an online support, a manual or technical documentation, I decided to build the next prototype of SmartFactory on Eclipse platform.

The Eclipse Platform is designed for building integrated development environments (IDEs) that can be used to create very diverse applications. This very important purpose of Eclipse and other reasons presented below determined me to migrate SmartFactory to it:

- it lets a programmer use several different tools from the same application. From the same front end, someone can combine tools for writing code with "plug-ins" for modelling databases or testing applications. IBM is using the Eclipse software to provide a common foundation for its suite of development tools, giving a disparate product set a common user interface as well as a mechanism to share information. This example was followed by all who joined Eclipse;
- it is a very fast growing project: very popular, with a lot of new tools, attracting many companies each year;
- it offers access to a huge community of researchers, students and companies in order to share and publish information.

Eclipse is a Software Development Kit (SDK) which, in metaphorical terms, may be compared to a machine lathe, where you cannot only *make products*, but also make the *tools for making the products*.

At the most fundamental level, Eclipse is actually the *Eclipse platform* whose purpose is to provide the services necessary *for integrating software development tools* as Eclipse *plug-ins*. The beauty of Eclipse's design is that, except of a small *runtime kernel*, everything is a *plug-in* or a *set of related plug-ins* (see Figure 25).

To be useful, the *Platform* has to be extended with plug-ins such as *JDT* (Java Development Toolkit) for writing, and debugging Java programs. In this form Eclipse is just a Java IDE (Integrated Development Environment) and the reality is that this is what most people use Eclipse for.

In order to build SmartFactory prototype I use Eclipse firstly to create my own plug-ins (see section 5.3) and then as the development continues to make use of *JDT* and other Eclipse tools to write and generate code.

Besides of the standard plug-in *JDT*, Eclipse contains *PDE* (Plug-in Development Environment) which makes Eclipse *easily extensible by other plug-ins*. The *Eclipse platform* consists of:

- a small *Platform* runtime kernel,
- *Workbench* (GUI: menus, toolbars, perspectives, views and editors),
- *Workspace* (to contain and manage *projects*),
- *Help* and
- *Team support* components.

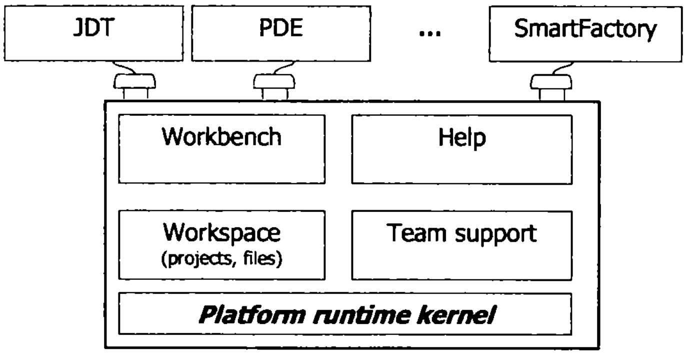


**Figure 25. The Eclipse Architecture**

The *Team support* plug-in facilitates the use of a *version control* (or configuration management) system. The Eclipse platform includes a client for *Concurrent Versions System* (CVS) – a source control tool which is very important if two or more people work together on a single set of files in order to track and coordinate changes.

Although Eclipse is written in Java, it is *language neutral* – additional plug-ins are available for other languages such as C/C++, Cobol, and UML. But it is not strongly *platform neutral* due to the decision to use the operating system's native graphics.

Eclipse framework is using a *workbench* which is a single application window that at any given time contains a perspective. A *perspective* is a pre-selected set of *panes* which also contain one or more tabbed *views* and each pane serves as an *editor* (see Figure 26).

The *Resource* perspective is considered as home perspective, a general purpose perspective for creating, viewing, and managing all types of resources. It contains, in particular, the *Navigator* view showing a hierarchical representation of the workspace and all the projects in it. Other perspectives are available via the main menu "Window -> Open Perspective" or via shortcut toolbar (perspective switcher). For developing SmartFactory prototype I preferred to utilize *Java* perspective because:

- it contains an useful *package explorer* view for visualizing the Java package and classes project structures:
- it understands Java packages
- it displays the packages as a single entry rather than as a nested set of directories;
- it contains *problems* (presenting the list of errors and warnings of the project) and *properties* (the list of the entities of a model properties and their values) views for building the prototype and
- it contains *console* and *error log* (the list of error messages generated by each plug-in) views for managing the plug-ins at run-time.

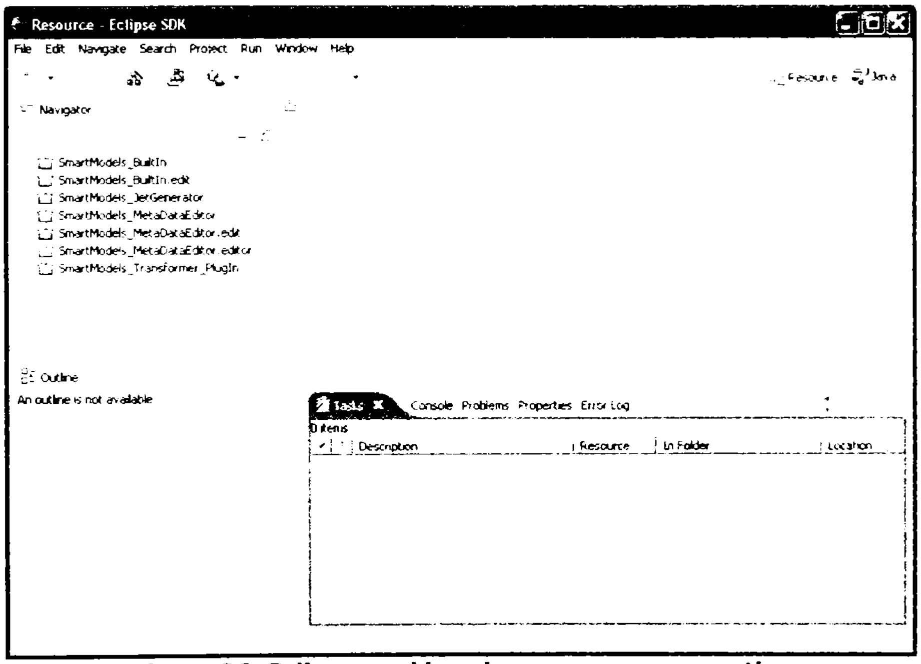

Resource - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Resource Java
Navigator
SmartModels_BuiltIn
SmartModels_BuiltIn.edit
SmartModels_JetGenerator
SmartModels_MetaDataEditor
SmartModels_MetaDataEditor.edit
SmartModels_MetaDataEditor.editor
SmartModels_Transformer_PlugIn
Outline
An outline is not available
Tasks Console Problems Properties Error Log
0 items
Description Resource In Folder Location

**Figure 26. Eclipse workbench – resource perspective**

The platform is very flexible in the way the perspectives can be changed:

- temporary super-sizing a view by double-clicking on the title bar;
- moving views around by dragging their title bars;
- closing a view;
- adding a new view: "Window -> Show View"
- restoring the perspective to its default appearance: "Window -> Reset Perspective";
- saving a customized perspective: "Window -> Save Perspective As".

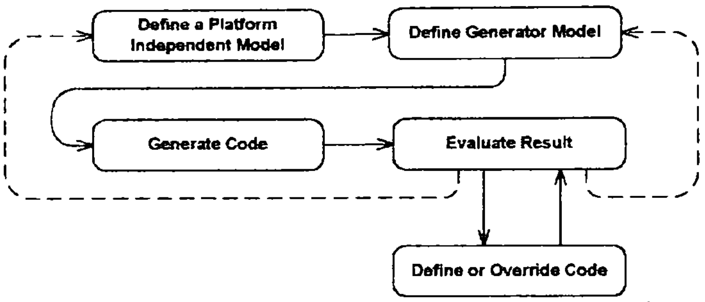


**Figure 27. Using EMF framework generative tools**

There are three standard types of projects in Eclipse:

- Java - the choice for developing a Java program (this is the type of project used for developing the Built-In and Meta-Data Editor - see sections 5.3.1 and 5.3.2);
- Simple - provides a generic environment to use for documentation (this type was not considered while developing SmartFactory);
- Plug-in Development - provides an environment for creating my own plug-ins to integrate into Eclipse (this is the type of project used for developing the Transformer - see section 5.3.3).

### 5.2.2. Eclipse Tools Project

This section introduces briefly the Eclipse main tools that I used to build (design and implementation) the second SmartFactory prototype. They were also very helpful to add value such as including a GUI for writing a model, automated code generation and automated creation of rich client applications.

The Eclipse Modelling Framework (EMF) [35] is an open source code generation tool (see Figure 27) distributed under the Eclipse Tools Project umbrella. It is designed following most of the OMG's Model Driven Architecture (MDA) [86] principles and it is considered to be a very good proof of concept for it.

EMF is capable of creating complex editors from abstract business models (OMG's PIM). These editors are implemented as plug-ins for Eclipse, so EMF is not a general-purpose code-generation tool. EMF creates feature complete implementations including persis-tence, business model implementation (OMG's PSM), editing framework and editors.

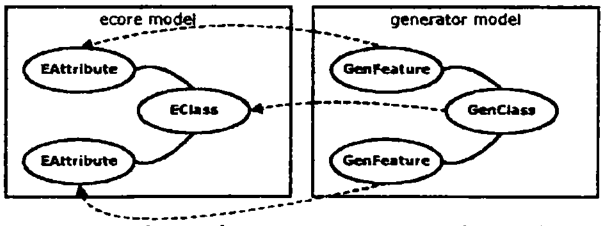


**Figure 28. The correspondence between Ecore model and Genmodel entities**

EMF has excellent support for an iterative development process, allowing the developers to change or fix the business-model or simply to change the code-generation options (in a model called *genmodel* - see Figure 28) and then regenerate the code. A user can specify which methods have been manually written, or they were automatically generated and have been updated by the programmer, and these methods will not be regenerated.

EMF provides the ability to save objects as XML documents for interchange with other tools and applications. These XML files have a ".*ecore*" extension and they will capture the business model requirements. EMF is also very flexible and lets the user define his *ecore* model in four ways (see Figure 29):

- Use a built-in primitive editor or other tools on the market that directly create *ecore* models (e.g., EclipseUML from Omondo - `http://www.omondo.com/`) using the UML notation. The EMF sample editor is actually an EMF generated editor, so EMF proves that it can be bootstrapped (it can be automatically generated by itself);
- Import from XML Schema;
- Import from Rational Rose Models;
- Import specially annotated Java Interfaces. This option is used for synchronizing the Java interfaces already coded or generated and manually changed by the user, with the *ecore* model and vice versa.

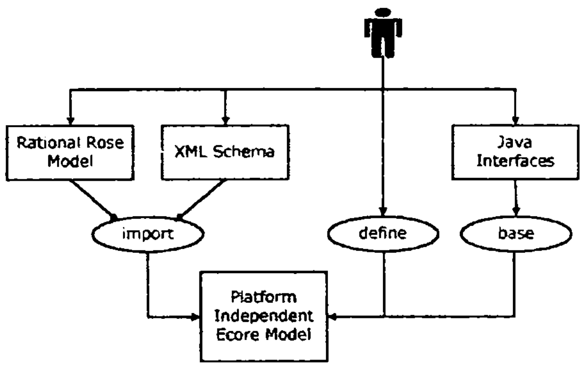


**Figure 29. Four ways to create an EMF Ecore model**

EMF includes an EMF-based implementation of Service Data Objects (SDO). SDO is a framework that simplifies and unifies data application development in a Service Oriented Architecture (SOA). It supports and integrates XML and incorporates J2EE patterns and best practices.

Graphical Editor Framework (GEF) is another important tool we can find in Eclipse Tools Project. It has been designed to allow editing of user data, generally referred to as *the model*, using graphical rather than textual format. In this way GEF is a very efficient assistant in building a visual editor for our entities of the business-models.

A very important aspect of visual editing with GEF is that it provides a default mechanism to react to user actions and mouse (mouse pressed or mouse dragged) or keyboard events. Another significant benefit of using GEF is the fact that it fully integrates with the Eclipse platform. Objects selected in the editor may provide properties for the standard *Properties view*. Eclipse wizards may be used to create and initialize models edited by GEF editors. In the generated editor menu a user can add user-defined *commands* which can modify the model. These commands will support by default undo/redo model alteration. *Undo* and *Redo* items of the *Edit* menu may trigger undoing or redoing of GEF editing changes.

A very powerful tool we can find in Eclipse-EMF is the possibility to generate code. The code generator reads the Ecore model, the generator models, and a set of code definition templates defined in a template language called Java Emitter Templates (JET - see Figure 30). JET plays a major role in the code generation process in EMF and it helps to understand better the EMF framework.

The JET template is not likely to be modified by most of the EMF users; however, in my second SmartFactory prototype I had to modify two of the templates in order that I could add in the generation process the semantic information of a business-model.

JET is based on the JavaServer Pages (JSP) syntax (in fact, the JET implementation was based on the Tomcat implementation). A JET template contains a template for files containing Java implementation files. The framework "expands" the various JET files to generate the Java source files.

Before going into SmartFactory implementation presentation I want to add one last paragraph about Eclipse Rich Client Platform (RCP) - a new proficient way to build Java applications. I used very much this feature in developing and testing my SmartModels business-model editor for second prototype of SmartFactory.

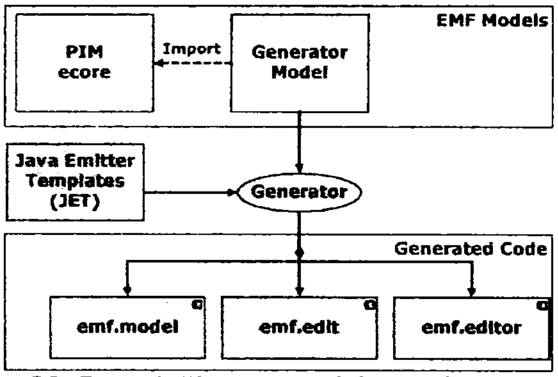


**Figure 30. From EMF Ecore model to code-generation**

Because of its unique open source license, RCP was designed to gain all the advantages of using all the technologies that went into Eclipse to create commercial quality programs. It saves the programmer of the heavy and trivial work to customize the menus, layouts, and other user interface elements. This is possible because of the "IDE-ness" of Eclipse was hard-wired into it (`http://www.eclipse.org/articles/Article-EMF-goes-RCP/rcp.html`) [35]. RCP, which is basically a refactoring of the fundamental parts of Eclipse's UI, allows it also to be used for non-IDE applications. Version 3.1 of Eclipse updated RCP with new capabilities, and, most importantly, new tooling support (new editor properties to support automatic generation of RCP from the EMF generator - model framework, new wizards) to make it easier to create plug-ins and applications than before. In this way an RCP application is able to manipulate instances of my EMF model - exactly as with ordinary EMF generated editors.

## 5.3. A New Design for SmartFactory Based on Eclipse

The new prototype of SmartFactory was developed on Eclipse platform. There are seven plug-in Java projects which work together to implement the SmartModels *mop*'s principles and rules. In the next sections I will present them with the role they have in the approach, the Eclipse features that they use and the design choices I have made in order to build them.

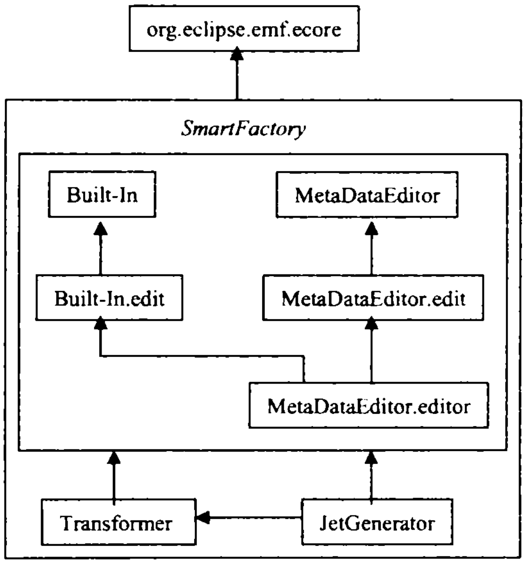


**Figure 31. SmartFactory Plug-ins**

Figure 31 presents the SmartFactory plug-ins architecture which highlights the links and the dependencies between them and Table 4 describes the role of each plug-in. From the very beginning it is important to observe that all the plug-ins make use of org.eclipse.emf.ecore tool plug-in:

- the Built-In and Meta-Data Editor use EMF Ecore both for their design and for implementation (see sections 5.3.1 and 5.3.2 for more information on the manner I customized them);
- the Code-Generator uses EMF.CodeGen for automation of the code-generation process for the SmartFactory transformed model (see section 5.3.4 for the manner I updated this tool in order to meet the requirements of my approach);
- the Transformer deals with EMF ecore model transformation. From the design point of view it did not really need to be described using ecore, but I still used it for reasons of unified development of the prototype and for conformance with the way EMF generated code for the other plug-ins.

**Table 4. Description of SmartFactory Plug-Ins**

| **Plug-in name (prefix: SmartModels_)** | **Description** |
|---|---|
| Built-In | - it is the kernel of SmartFactory prototype;<br>- it implements the Meta-Object Protocol approach;<br>- it reifies the SmartModels entities. |
| Built-In.edit | - it contains EMF content provider classes to describe entities using the editor. |
| MetaDataEditor | - it represents the SmartFactory Meta-Data Editor;<br>- it customizes the EMF ecore entities in order to support the SmartModels entities specific properties;<br>- it implements the SmartModels methodology to describe a model. |
| MetaDataEditor.edit | - it contains EMF content provider classes to describe the Meta-Data Editor specific entities for the editor. |
| MetaDataEditor.editor | - it is the GUI of the Meta-Data Editor;<br>- it provides a wizard and EMF panes to edit a SmartModels model;<br>- it is an Eclipse RCP application. |
| Transformer | - it performs a model transformation from the Meta-Data Editor format to an EMF ecore format so I can reuse the EMF.CodeGen to leverage the code-generation process;<br>- it uses annotations containing Java pure code to add the SmartModels approach value to EMF entities. |
| JetGenerator | - it updates the EMF.CodeGen to take into account, when generating code, the SmartModels specific annotations attached by the Transformer to the EMF ecore model. |

### 5.3.1. The Built-In Plug-In

The Built-In plug-in represents the starting point of the implementation of SmartFactory. It is the reification of all the entities of SmartModels and can be distributed as a library jar file called *"SmartModels_BuiltIn.jar"*. It was developed using EMF Ecore. In order to add the Meta-Object Protocol (*mop*) approach to it, a part of the code of this plug-in is manually written by me. In the next paragraphs I will explain how I used EMF to implement each entity quoting samples from the *ecore* file.

The Built-In plug-in has four main packages. The *mop package* includes the classes used to implement the *mop* policy. The root of all the SmartModels objects – any SmartModels entity which has relevant *mop* information for the approach – is *SmartModelsObject* (see [106] [108]). This information is manually added only in this plug-in and it contains Java class *"Class"* objects of the current entity, its super-class (because all this entities are first-class elements of SmartModels) and its concept (its semantic information).

The content of a SmartModels *mop* object is very similar with the previous SmartFactory prototype built on SmartTools platform. I imported most of the code, except what is automatically generated by Eclipse EMF *genmodel* tool. Here is an extract of the model object for *mop* from the Built-In ecore file:

```
<eSubpackages name="mop" nsPrefix="org.smartmodels.mop">
- <eClassifiers xsi:type="ecore:EClass" name="SmartModels_MOP"
eSuperTypes="#//smartmodels/mop/SmartModelsObject">
+ <eOperations name="addIntoExtent">
<eParameters name="javaClass" eType="ecore:EJavaClass"/> …
+ <eOperations name="addIntoExtentNotRecursive">
<eParameters name="javaClass" eType="ecore:EJavaClass"/> …
+ <eOperations name="addIntoExtentRecursive">
<eParameters name="javaClass" eType="ecore:EJavaClass"/> …
+ <eStructuralFeatures xsi:type="ecore:EAttribute"
name="descriptionName" eType="ecore:EString"/>
+ <eStructuralFeatures xsi:type="ecore:EAttribute"
name="metaDescriptionName" eType="ecore:EString"/>
+ <eStructuralFeatures xsi:type="ecore:EAttribute"
name="specialisedDescriptionName" eType="ecore:EString"/>
+ <eStructuralFeatures xsi:type="ecore:EAttribute"
name="descriptionClass" eType="ecore:EJavaClass"/>
+ <eStructuralFeatures xsi:type="ecore:EAttribute"
name="metaDescriptionClass" eType="ecore:EJavaClass"/>
+ <eStructuralFeatures xsi:type="ecore:EAttribute"
name="specialisedDescriptionClass" eType="ecore:EJavaClass"/>
+ <eStructuralFeatures xsi:type="ecore:EAttribute"
name="extent" upperBound="-1" eType="ecore:EJavaClass">
</eStructuralFeatures>
</eClassifiers>
…
</eSubpackages>
```

*mop* information is implemented as a Java static code and it automatically and recursively updates the SmartModels database of *mop*s. Because of that I decided not to let the *mop package generated* MopFactory to create instances of SmartModels_MOP objects. By default, EMF generates a factory for each package from the business-model *ecore* file. In my approach the creation of *mop objects* is exclusively the job of the database and it provides a full set of methods in order that it can be queried by the meta-programmer. Another consequence of having a database of *mop*s is the need to design the *mop* containing class as a singleton. EMF Ecore does not provide the possibility to define singleton classes, so I had to forbid the MopFactory to create SmartModels_Mop DataBase instances.

Therefore, I chose to more clearly separate the *mop* policy from the other entities. That is why I decided to build a database for *mop* instead of having the atom *Atom*, the root of SmartFactory first prototype, to act as a database.

It is also important to remember that in order to use the *mop* facility the meta-programmer needs to initialize the SmartModels database by adding the root of the concepts' singleton instance (for example, the lines of code below are used when I launch the MetaDataEditor for editing SmartModels business-models. In order to edit entities which have a *mop* from the Built-In plug-in or which inherit from them, I need to initialize the *mop* database):

```
public static void initSmartModels(){
      ConceptImpl.getSingle();
}
```

If this object is not found in the database, then it will generate a java.lang. ClassNotFoundException when it tries to load the java.lang.Class object for a particular entity which has *mop* information.

The *core package* is actually the reification of the SmartModels first-class entities (see section 3.3). Its structure reflects the evolving SmartModels architecture and aims to facilitate the reusability. That is why some of the properties are missing (compared to the previous SmartFactory prototype), in order that I can reuse all the content of the Built-In plug-in in the Meta-Data Editor and also that I can better specify these properties in the editor so it can take full advantage of the EMF Java code generation based on the model.

The *core package* includes all the SmartModels basic entities even if some of them are not yet fully defined, but the prototype offers a platform flexible for future development. It is organized as a list of sub-packages identifying the different element customizations (some of them are imported from OFL [24]):

- *atoms package* – an implementation of the reification-level in SmartModels. The organization of this package was determined by the two axes of co-ordinates which I use to distinguish between atoms: the support for generic entities and the support for atoms which have instances within applications;
- *concepts package* – similar approach for implementation like the *atoms package*;
- *behaviors package* – identifying actions, aspects and visit-entities;
- *customizations package* – implementing the SmartModels root for meta-definitions (hyper-generic parameters and characteristics definitions) and the root for meta-values (hyper-generic parameters and characteristics values) for modeling the semantics of a business-model. Customization, behaviors and concepts packages form the implementation of the meta-level in SmartModels;
- *facets package* – for modeling the application-level in SmartModels (the facets are not yet fully defined);
- *assertions package* – for implementing the OCL constraints [82] (this is currently under investigation on how to describe and use them, therefore it is not fully implemented in the current version of SmartFactory).

A new interesting improvement I made in this prototype is the direct access of each entity which has *mop* information to its corresponding concept. This is implemented through a method which is redefined in each class in order to return the correct sub-type of concept thanks to the new feature in J2SE 5.0 that allows covariant return types. This means that a method in a sub-class may return an object whose type is a sub-class of the type returned by the method with the same signature in the super-class. In this way, for each entity one can know exactly what the corresponding concept is and manipulate the semantic information (query the parameters/characteristics or execute actions):

```
public GenericConcept getAccessToConcept(){
        return (GenericConcept)getConcept(GenericAtomImpl.class);
}
```

This method finds the single instance of the attached concept of an entity (for a *GenericAtom* in the code example) by delegating the searching process to the SmartModels mop database, each object providing its Java Class object as a parameter. This is automatically generated for all SmartModels entities, also from the other SmartFactory plug-ins or for plug-ins generated by users who create their own business-models (see section 5.3.4).

Concepts are designed as singleton and this is an important issue of the SmartFactory prototype implementation. I want all the entities which share the same semantic information to have access to the same instance which holds this information that is their concept. Because genericity in Java does not support the SmartModels meta-information approach (parameters/characteristics, actions, etc.), but I want to reuse as much as possible the generative programming from EMF, I decided not to design concepts as abstract and static classes anymore (as they were in the first prototype), but as EMF Java generated classes adding the singleton design pattern.

The design of parameters and characteristics in the *customization package* addresses important concerns in SmartFactory. First of all, I distinguish between the meta-definition of a parameter and its meta-value. The definition will specify the type of a parameter (see next paragraphs on SmartModels types) and a default value. The meta-value will have a reference to the definition to which it conforms and the SmartModels value.

The following lines are the parameter representation from the built-in *ecore* model and the design is identical for characteristics:

```
- <eSubpackages name="parameters" nsPrefix="org.smartmodels.core.
customizations.parameters">
- <eClassifiers xsi:type="ecore:EClass" name="SmartModels_HyperGeneric
      ParameterDefinition" eSuperTypes="#//smartmodels/core/
      customizations/SmartModels_MetaDefinition">
- <eOperations name="getType" eType="#//smartmodels/util/types/
SmartModels_TypeForHyperGenericParameters"> ...
- <eOperations name="getDefaultValue" eType="#//smartmodels/
util/values/SmartModels_ValueForHyperGenericParameters"> ...
</eClassifiers>
- <eClassifiers xsi:type="ecore:EClass" name="SmartModels_HyperGeneric
ParameterValue" eSuperTypes="#//smartmodels/core/customizations/
Smartmodels_MetaValue">
<eStructuralFeatures xsi:type="ecore:EReference" name="definition"
lowerBound="1" eType="#//smartmodels/core/customizations/
parameters/SmartModels_HyperGenericParameterDefinition"/>
<eStructuralFeatures xsi:type="ecore:EReference" name="value"
eType="#//smartmodels/util/values/SmartModels_Value"
containment="true"/>
</eClassifiers>
</eSubpackages>
```

This shows that in the meta-definition I did not define the type and the default value of the parameter as attributes, but just as accessors to them knowing that each concrete parameter will define them. This is because Eclipse EMF does not have covariance for the attribute types and in case of a redefinition of an attribute it produces errors in the generated Java code.

SmartModels Meta-Value has also a fine detail in the design of the references it defines. I decided to set the attribute *containment* to true for the value reference and not for the definition reference. This is because the meta-value will create an internal SmartModels value, but the definition is just a reference to a meta-definition created in the parameter customization description. A value cannot contain the definition (I do not create the definition whenever I create a value, but I can create multiple possible values for the same definition). In the design of the Meta-Data Editor I heavily used the possible settings for containment attribute and there will be more examples on this issue (see section 5.3.2).

The *util package* contains all the SmartModels built-in entities which do not have a relevant *mop* information, but contribute to the modelling of the semantic information. It is structured as four sub-packages which clearly identify:

- *exceptions* – a set of SmartModels specific *exceptions*. This is an open list where I will continue to define throwable exceptions in order to achieve a better understanding and specification of the detail messages describing the conditions of executing a SmartModels application. I will point out several exceptions in the contexts that they may be triggered;
- *types* (see Figure 32) – contains the SmartModels type system which is inspired from OCL basic types [82], from UML model and from MOF [84] architecture;
- *values* (see Figure 33) – contains the SmartModels value system which mirrors the type system;
- *redefinitions* – contains the SmartModels possible value redefinitions used in the reification of the derived atoms.

SmartModels provides a set of primitive types (Boolean, Integer, Real, String) which conform with the OCL standard library [82] where they are all defined as instances of the Primitive meta-class from the UML core package. There is also a set of collection-related types from the context of UML. Each one of them has a correspondent type in OCL expressions. In the SmartModels approach I classified them as *Enumeration*, *Collection* and *Tuple* (see next paragraphs). *Tuple* type was defined from the need of capturing semantic information for entities whose properties are defined as an Atom or Concept.

Section 3.4.2 mentioned some parameter and characteristic type examples and in the business-model example from next chapter I will give several other examples. There is also an important implementation discussion in section 5.3.2 on the reason why I decided to design the SmartModels Meta-Data Editor in order to be able to use *types* package modelling (using UML) and code-generation from Eclipse EMF and conform to the J2SE 5.0 which provides multiple inheritance for interfaces, but not for classes.

The root of the SmartModels type system is *SmartModels_Type* class. Figure 32 shows a clear distinction between the parameter types and characteristic types. They are both designed to capture the semantics of the SmartModels entities, but parameters will encapsulate basic type properties, while characteristics will describe more complex meta-information.

The parameter types include the *primitive types* and what I called the *predefined types* (Enumeration, Collection and Tuple). The characteristic types are reified also by the predefined types and by what I called a *user-defined type* (SmartModels Atom or Concept type). However, Figure 32 is not very clear (it does not aim this due to the its large size anyway) in making the difference between the way parameters use the predefined types compared to the characteristics. Indeed, for a parameter the template of the element types for a SmartModels *Collection* or *Tuple* type can be only a primitive type, while for a characteristic will be another predefined or a user-defined type element.

As I mentioned above both the SmartModels Collection and Tuple are template types. The Collection type has one parameter and Tuple has two or more. For both of them this element type is of SmartModels_Type, the root of the type system. This means I allow to recursively define complex types: i.e. a tuple type with one or more of the element types being a collection, then a collection having the template parameter substituted with a tuple type and so on.

In order to have a flexible description of the Collection type and to conform with the OCL standard library [82] I added two more boolean attributes (*isOrdered* – if the elements are in sequence order; and *isUnique* – if one object can be an element only one time, duplicates are not allowed) and two positive integer attributes (*minimalCardinality/ maximalCardinality* – the minimum / maximum number of elements in the collection).

By default minimal cardinality is set to 0 and maximal cardinality is set to java.lang.Long.MAX_VALUE and this means there are no constraints on the size of the collection. If minimal cardinality has a positive value then SmartModels forbids removing elements if the size of the collection is inferior or equal to that value. Adding an element may be also forbidden if the size of the collection is already equal or bigger than maximal cardinality.

The Collection type is also equipped with many operations which help the programmer to manipulate them in an easy and flexible way. It enables the construction of new collections from other existing ones or from arrays. It provides operations to add/insert/remove elements while verifying the constraints, to check the existence of an object in the collection, to iterate over it, to easy access the elements.

Figure 32 presents the *Tuple* type with a reference called *elementTypes* with cardinality of 2..* to SmartModels_Type (these are the template's parameters) and with a positive integer attribute called *nrOfElements* representing the number of items contained in *elementTypes*. This attribute has the *volatile* EMF modifier which means that at the code-generation time *nrOfElements* will not be a concrete attribute, but a method which will compute this number and will return it each time it is queried. This is an interesting feature from the point of view of the Meta-Data Editor (see section 5.3.2).

The *getDefaultValue* method of the SmartModels_Type root is redefined in each concrete type with the corresponding return type. In fact, in the root interface I just defined an abstract method, while in the heirs there are references to the analogous SmartModels_Value type and EMF code-generator automatically generates accessors for them.

The *isValidRedefinitionType* is different from *getDefaultValue* because this method is strictly redefined in the SmartModels_Type' heirs. This method indicates if a redefinition (see next paragraphs) of a value of the related type is valid or not.

The *isValid* method checks if a value conforms with a type. This is not redefined in the SmartModels_Type' heirs, but it delegates the verification to the equivalent method *isConsistentWithType* from the SmartModels_-Value (see next paragraphs).

The association between SmartModels concrete types and Java language is reified through manually written *getJavaName* methods which return as a java.lang.String the name (if there is a direct correspondence with a Java basic type) or the Java code that reifies it. This method is used in the *SmartModels-Transformer* plug-in (see section 5.3.3) to drive the code-generating process (see section 5.3.4).

The *SmartModels values* are organized in a hierarchy parallel to the types as I can see in the diagrams of Figure 32 and Figure 33. In this way I can very easily check the conformance of the values with the corresponding types. The root of the values system is *SmartModels_Value* interface.

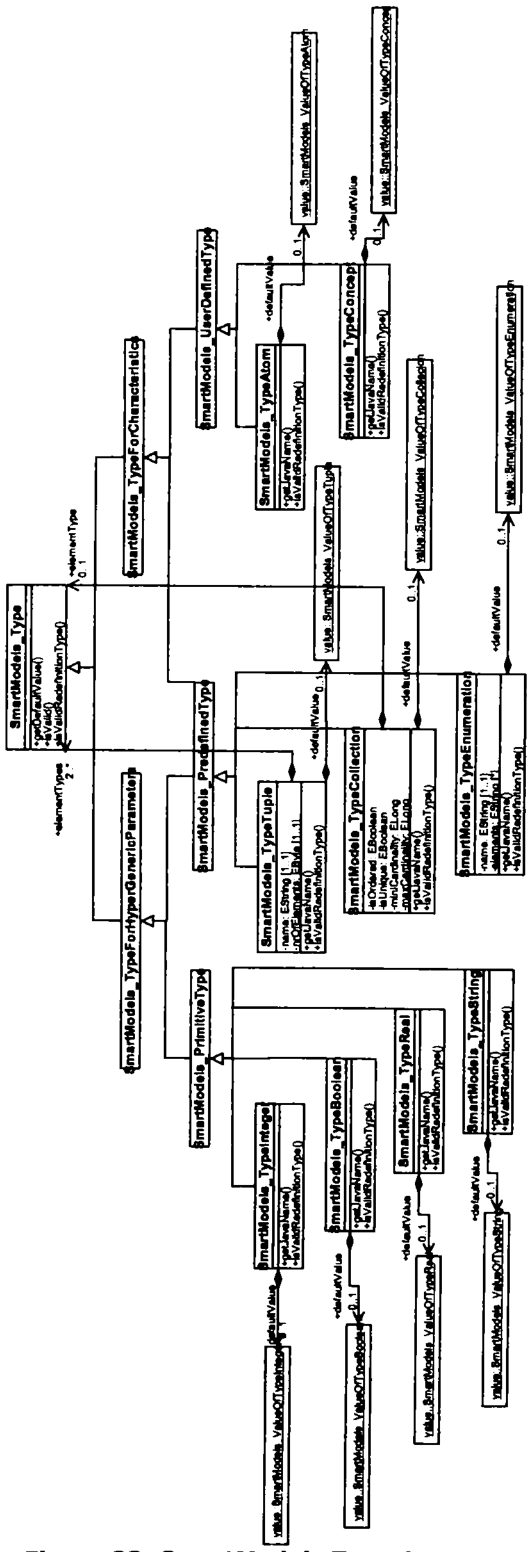


**Figure 32. SmartModels Type System**

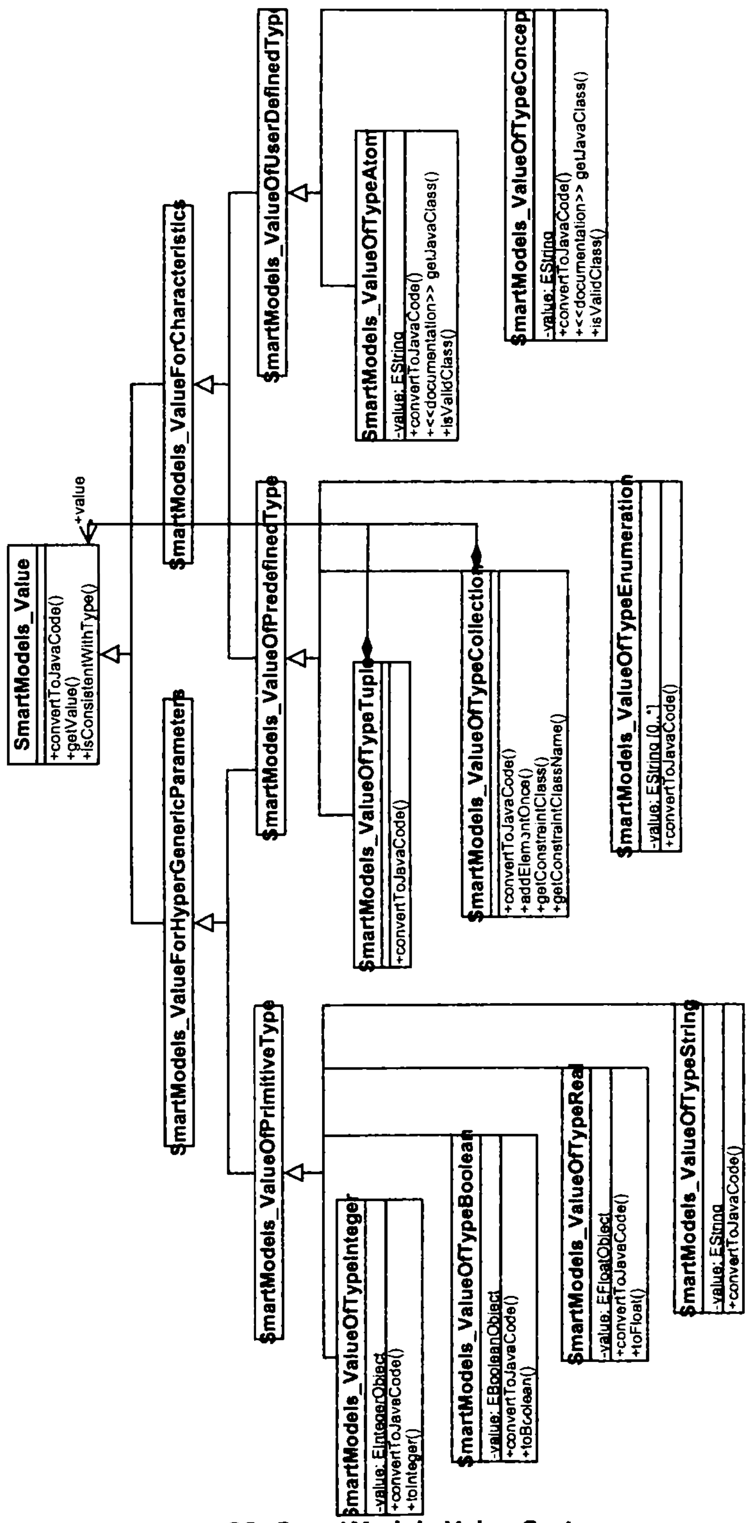

SmartModels_Value
+convertToJavaCode()
+getValue()
+isConsistentWithType()
+value
SmartModels_ValueForCharacteristics
SmartModels_ValueForHyperGenericParameters
SmartModels_ValueOfUserDefinedType
SmartModels_ValueOfPredefinedType
SmartModels_ValueOfPrimitiveType
SmartModels_ValueOfTypeAtom
-value: EString
+convertToJavaCode()
+<<documentation>> getJavaClass()
+isValidClass()
SmartModels_ValueOfTypeConcept
-value: EString
+convertToJavaCode()
+<<documentation>> getJavaClass()
+isValidClass()
SmartModels_ValueOfTypeTuple
+convertToJavaCode()
SmartModels_ValueOfTypeCollection
+convertToJavaCode()
+addElementOnce()
+getConstraintClass()
+getConstraintClassName()
SmartModels_ValueOfTypeEnumeration
-value: EString [0..*]
+convertToJavaCode()
SmartModels_ValueOfTypeInteger
-value: EIntegerObject
+convertToJavaCode()
+toInteger()
SmartModels_ValueOfTypeBoolean
-value: EBooleanObject
+convertToJavaCode()
+toBoolean()
SmartModels_ValueOfTypeReal
-value: EFloatObject
+convertToJavaCode()
+toFloat()
SmartModels_ValueOfTypeString
-value: EString
+convertToJavaCode()

**Figure 33. SmartModels Value System**

Then there is a clear distinction between the possible parameter and characteristic values taking into account the established fact that a parameter will describe basic properties of an entity's semantic information and a characteristic expresses a property whose value is defined by an atom/concept or a set of atoms/concepts (enumeration, tuple or collection). This means that what I called *SmartModels_ValueOfPrimitiveType* is exclusively a possible value for parameters and *SmartModels_ValueOf-UserDefinedType* is for characteristics.

It is again important to understand something which is not visible from the diagram. A parameter value can be assigned with a *SmartModels_Value-OfPredefinedType* only containing values of primitive type, but a characteristic will be assigned with values of Atom/Concept types. This remark is a reflection from the same policy I have for types.

The values of the primitive types have a straightforward Java implementation. They are Boolean (boolean in Java), Integer (int), Real (float) and String (String). The value of an Enumeration is also a java.lang.String. The template parameters of a Collection or Tuple can be instantiated with any SmartModels_Value objects. The difference consists in the fact that in a Tuple value I can compose several SmartModels values each of which has a name and can have a distinct type.

The *getValue* method returns the value stored in a SmartModels value. It is defined in the root of the hierarchy with a java.lang.Object return type and each concrete SmartModels value holder will redefine it. The values of the primitive types are equipped with methods to return the correspondent Java value wrapper. (i.e. SmartModels_ValueOfType-Boolean defines a method called *toBoolean* which returns a java.lang. Boolean object).

The *isConsistentWithType* method is the equivalent method of the *isValid* from the SmartModels_Type. Here it is strictly redefined in the heirs where the SmartModels_Type just delegates to this method the verification of the conformance of a value with a type.

The *convertToJavaCode* method is a cross-functional operation used only by the *SmartModels_Transformer* plug-in (see section 5.3.3) and it was defined to help the code-generation process for SmartModels values (to translate them into pure Java language value system).

There is an interesting approach in the implementation of the values for Atom/Concept types. Because not all Atom/Concept entities of a SmartModels business-model have instances, I could not integrate them with the other values (which all have at least a java.lang.Object value). But I also need to take into account that all I need from this types of values is the semantic information and this is stored in the singleton instances of the concepts. That is why I decided to represent an atom/concept value as a java.lang.Class object. Therefore I can integrate them in the SmartModels value hierarchy and a programmer can have easy access to the meta-information (there are also other advantages in the future design of the Meta-Data Editor – see section 5.3.2).

The values of Collection and Tuple types are equipped with a set of specific operations. Among them there are methods like *getConstraintClass* that returns the java.lang.Class<T> to which elements of the collection are constrained. This also triggers a *ConsistencyException* if the result is false.

In the SmartModels approach a *Derived Atom* can be obtained in two ways:

- through setting the values of the parameters and characteristics which compound the semantic information of its Generic Atom; or
- through redefining the values of the parameters and characteristics of another Derived Atom which relies on exactly the same Generic Atom.

Knowing that, in the next few paragraphs I will introduce the notion of value redefinition in SmartModels. The last sub-package from the SmartModels *util package* is the *redefinitions package* and it contains all the possible redefinitions.

The list of the possible redefinitions resides on the values architecture and most of them can apply only to one type of value. The redefinition that imposes no constraint is *SmartModels_RedefinitionFree* and this means that any value which conforms with the corresponding type is allowed; if this condition is fulfilled, then the redefinition value is always valid.

There is a set of redefinitions that applies to values of primitive types like: *Lower/Greater* (and they depend on the implementation languages operators) or a set which have sense in the context of values of type collection like: *BiggerCollection/SmallerCollection*. *SmartModels_RedefinitionBiggerCollection* is the redefinition described by a collection of values which can only have the same list of elements like the original collection and maybe some more (it is not just size that matters). *SmartModels_Redefinitions SmallerCollection* is the redefinition described by a collection which can only have a smaller list of elements from the original list. Like in the case of the bigger-collection constraint, it is not just the size which matters (now it is smaller), but from the original list some of the elements cannot occur anymore.

The *SmartModels_RedefinitionsPairs* is a type of value redefinition which applies to all types of values. It contains a list of valid redefinitions of type *SmartModels_RedefinitionPair*. This is a part of a redefinition which has a pair of SmartModels_Value: the old and new values. In most of the real world cases it is very useful to define for each old value to which new value it can be redefined. This is not fully implemented in the new SmartFactory prototype mostly because of the complexity it involves in the design of the Meta-Data Editor, but there are a couple of examples in the business-model presented in next chapter.

Section 5.2 presented the fact that Eclipse EMF supports three levels of code-generation. I used the first level for generating the SmartModels_Built-In plug-in that I described in the paragraphs above. The meta-model (an ".ecore" file) from the core EMF framework describes business-models and at generation time it provides Java interfaces and implementation classes for all the classes in the model, plus a factory and package (meta data) implementation class. I also presented the adaptations that I manually made in the code in order to implement the SmartModels principles and rules approach.

The second level of code-generation in EMF generates implementation classes (called *ItemProviders*) that adapt the model classes (adapters) for editing and display. They form a new plug-in with the same name and the ".edit" (***SmartModels_BuiltIn.edit***) suffix and it depends on the first plug-in. It includes generic reusable classes for building editors for EMF models [35] which provide:

- content and label provider classes, property source support, and other convenience classes that allow EMF models to be displayed using standard desktop viewers and property sheets;
- a command framework, including a set of generic command implementation classes for building editors that support fully automatic undo and redo.

The third level of the EMF code generation facility is capable of generating everything needed to build a complete editor for an EMF model. It includes a GUI from which generation options can be specified, and generators can be invoked. The generation facility leverages the JDT (Java Development Tooling) component of Eclipse. It produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors and serves as a starting point from which to customize the meta-model. This constitutes the third plug-in with the same name as the model and the ".editor" suffix and it depends on the previous plug-ins as it is presented in the Figure 31.

The kernel of the SmartFactory cannot be edited (that is why I chose to call it "Built-In") and I did not generate the editor plug-in for it. But the edit plug-in of the SmartModels_Bult-In exists because it is used by the Meta-Data Editor (see section 5.3.2) which relies on the Smart-Models basic entities.

### 5.3.2. The Meta-Data Editor Plug-Ins

The SmartModels_Built-In plug-in encapsulates the reification of the SmartModels entities. It produces a set of Java classes for its core model and section 5.3.1 presented the changes that were made in the code to adapt it to the SmartModels principles and rules.

Now the prototype needs to provide also an editor in order to be able to easily create business-models and then write applications. To accomplish this objective I use the powerful and open-source EMF tools.

This section presents the design of the Meta-Data Editor with respect to the SmartModels requirements and adapted to the EMF limitations. Section 5.3.3 presents the transformations I performed on the editor output to construct a new customized *ecore* file representing our model. Then the last step in order to generate a pure Java code, but which reflects the SmartModels approach, is described in section 5.3.4 where I customized the EMF.Codegen plug-in.

The heart of the *SmartModels_MetaDataEditor* plug-in is again an EMF *ecore* model file. For the reason that using just the EMF framework (its UML diagram editor) I do not have all the tools to describe all the SmartModels entities I needed to create my own editor. In order to realize that, I enriched the sample EMF *ecore* editor with support for SmartModels entities.

Before shortly presenting the editor's components it is important to review the SmartModels methodology to describe a business model presented in section 3.4.5.

Therefore I created the *SmartModelsEditor* which is the root of the editor and acts as a database holder of SmartModels entity reifications. It is the container of these three lists and has only one constraint: the list of Atoms cannot be empty. The root implements the org.eclipse.emf. ecore.ENamedElement interface [36] and this means it inherits all the properties of an EMF EObject and I also can add annotations and the name of the model.

For each of the three main entities of my editor I created a correspondent model object and I named them after the original entities adding the suffix "*Editor*". They all have org.eclipse.emf.ecore.EClass as a super-type and this choice has many advantages:

- it benefits from the EMF (UML oriented) framework which is currently under a rapid development and it is more than likely that there will be richer models in the future;
- it saves a large amount of work to build a representation of the entities of an object-oriented programming language (OOPL);
- it draws the modelling phase closer to the implementation language (which has to be an OOPL);
- it helps the programmer to freely add other attributes, references or methods which may allow him describe better the model entities.

The *meta_model package* is an editor extension where I put together the SmartModels core entities customizations. It includes adapters for atoms, concepts and derived atoms, actions, types and values for hyper-generic parameters and characteristics. In the following paragraphs I will briefly present them.

Java language does not provide the notion of meta-class and for that reason *AtomEditor* defines a reference to its meta-information possessor, which is its concept. The concept of a BasicAtom is BasicConcept or one of its heirs from the business-model or from another imported model (see section 5.3.1). EMF editors provide this facility (importation of resources) and one of the future goals is to investigate how to compose several SmartModels business-models. Analogous, the concept of a GenericAtom may be one of the concepts from the GenericConcept hierarchy.

*ConceptEditor* is the editor customization of the root of concepts from SmartModels. One of their powerful properties is that through actions they can dynamically control the behaviour of atoms in an application according to their semantic information and the set of the associated assertions (see section 5.3.1).

*ActionEditor* reifies the SmartModels notion of an action. Besides the important advantages we have seen for the atoms and concepts editor entities, I also designed ActionEditor as an EMF model object of type org.eclipse.emf.ecore.EOperation because it has to encapsulate the execution (operation) which can be performed by that action. This means that I can add UML value to an action by having the platform specify its containing class (its concept), a list of parameters and a return type, and exceptions that may occur.

In addition to that and in order to enhance the expressiveness of an action a user may build up three more lists in the editor:

- the list of other actions from the same concept which may be called by the current action. Each one of them can be stand-alone actions, but also can become parts of the current action if it needs them to execute their particular semantic behavior. This is a good mechanism to underline the dependencies between the actions of a concept;
- the lists of hyper-generic parameters and the list of characteristics that are interesting for the current action. An action may not query all the parameters of a concept, thus through these lists a user can highlight only the part from the meta-information of its concept which makes sense for this action.

As we have seen in section 5.3.1, actions are *first-class* entities and consequently the SmartFactory code-generator will create a distinct class for each one of them and will link them automatically to their concept and vice-versa, the concepts to the related actions. In this way I set the skeleton for future development of the actions. Furthermore, a practical application of the three lists mentioned above is that SmartFactory will generate accessors for each of their elements.

The next essential feature of the ConceptEditor is the capability to specify the parameters and characteristics. They are relevant and can be specified only at the level of a GenericConcept. A restriction I set in the meta editor is the lower limit of the number of parameters. It will accept the definition of a GenericConcept only if it has at least one parameter/characteristic defined. Otherwise, there is no reason to be generic, but basic concept. There is no upper limit for the number of parameters a user can define in a concept, but this is given by the object-oriented engineering principles for encapsulation.

The editor provides two entities in order to define meta-information. The commonalities between them are described in an org.eclipse.emf.ecore.

ENamedElement called *SmartModels_MetaDefinitionCreator* which is the abstract root of all entities of the editor which handle the creation of the parameters and characteristics. Thus, it declares accessors for the type, the default value and the possible value redefinition of a parameter.

This data will be stored in the heirs with the right type for each field and as a consequence they will automatically implement the accessors defined in the root. I followed here the same policy like for types and values in the Built-In plug-in (see section 5.3.1). *SmartModels_HyperGenericParameterDefinitionCreator* represents the root of all the editor parameter creators and in the same way I have the corresponding *SmartModels_CharacteristicDefinitionCreator* in order to create characteristics. More information about this hierarchy will be given in the next paragraphs when I will approach the architecture of the *util* package in the editor.

The Meta-Definition Creator also defines two methods which are not abstract and which will not be redefined in the heirs. They both test the value conformance with the type:

- *isValidValue* checks if a SmartModels_Value conforms with the type of the related parameter;
- *isValidRedefinitionType* checks if a SmartModels_Redefinition conforms with the type of the possible value redefinition of the related parameter.

The Meta-Value Creator is designed very similar to the Meta-Definition Creator. It represents the root of the SmartModels_Value creators in the Meta-Data Editor and it also declares two abstract methods: to access the definition of parameters and characteristics; and to access the appropriate SmartModels_Value wrapped in this meta-value creator. This methods will be implemented in the concrete heirs (see more information about value creators in the following paragraphs describing the editor *util* package) with the right return types.

The value creator hierarchy continues again with two abstract classes which have only an architectural role to distinguish between the creators of the parameters (*SmartModels_HyperGenericParameterValueCreator*) and the characteristics (*SmartModels_CharacteristicValueCreator*) values.

In the meta-model package of the editor we were introduced to the roots of the types and values creators. The *util* package of the editor contains the complete hierarchies of the creators of types, values and redefinitions. They were designed in order to maximize the use of the EMF generated editor. One of its most important characteristic is that in order to describe a reference of an object it automatically browses down the hierarchy of the type of the related reference and fills the context menu with all the concrete types which it can find through polymorphism.

I made use heavily of this characteristic and defined all the possible combinations of meta-definition types creators. It is true that I duplicated much data, but I achieved a much better control of the editor. That is why, for example, there is a *SmartModels_HyperGenericParameterDefinitionCreatorOfTuple* and also a *SmartModels_CharacteristicDefinitionCreatorOfTuple*. They both hold the same data, but from the *GenericConceptEditor* point of view there is a clear distinction between parameters and characteristics creators.

Nevertheless, it is also important to mention that I still need to add more control to the editor apart of what I could state through polymorphism. For example, a parameter of type tuple can contain only elements of primitive types, while the elements of a characteristic of type tuple can be of user-defined or predefined type. In the next paragraphs I will explain why I can impose a complete control on collections and not on tuple type creators using the polymorphism in the Meta-Data Editor design.

The design of the type creators in the Meta-Data Editor presents two important decisions I made in order to better express the SmartModels types architecture: the first one is about the reification of the collection creators and the second one is about the reification of the user-defined type creators.

For the same reason of organizing the creators of the SmartModels types in accordance with to their architecture in the Built-In plug-in and to better control them in the EMF editor, the *util* package contains a whole sub-package of possible collection type creators. Collection in SmartModels is a template type (it conforms to OCL [82]). Analogous to the tuple type, the template of a parameter of type collection can be instantiated only with primitive types, but for a characteristic it can be a predefined or a user-defined type. This means that the number of possible instantiations of the collection template is a finite number, but for a tuple type which can have an unlimited number of element types, this number is infinite. That is why the editor can control the creation of collections in contrast with the creation of tuples.

*SmartModels_DefinitionCreatorOfCollectionOfBoolean* is an example of a collection creator which contains Boolean objects (the template is of type SmartModels_ TypeBoolean). In this context it is obvious that the *defaultValue* reference type is of the corresponding wrapper of collection of Boolean values and the possible *redefinition* values reference is the editor reification of collection possible redefinition types for Boolean.

When I designed the user-defined type creators I had to take into account that the editor has its own dedicated entities for describing the Atoms and Concepts and thus I could not reference directly from the editor the SmartModels corresponding entities for building these types. That is what determined me to create in the editor two new entities for describing this types: *SmartModels_TypeAtomEditor* and *SmartModels_Type ConceptEditor*.

They are just substitutes in the editor for the original Atom and Concept types from the Built-In plug-in and before the code generation phase the transformer (see section 5.3.3) will replace them with the appropriate *SmartModels_UserDefinedTypes*. A straightforward use of this new type surrogates in the editor is that they are referenced from the *SmartModels_Characteristic DefinitionCreatorOfAtom* and from *SmartModels_DefinitionCreatorOfCollection OfAtom* (also for Concept).

The SmartModels value creators are symmetric to the type creators. The design choices I have made were derived from the same reasons like for types and also to conform with the type architecture. This applies also for both decisions I have made for the types architecture: the editor reifies all the possible instantiations of the collection template values and it defines *SmartModels_ValueOfType AtomEditor* (also for Concept) in order to describe a user-defined value. For tuple values I will need to add more control in the editor.

Each value creator from the editor contains two references:

- *value* which holds the adequate SmartModels value (e.g., there is *SmartModels_HyperGenericParameterBooleanValueCreator* which holds a Boolean value);
- *definition* which points to the editor type definition creator (e.g., the definition creator of type Boolean).

The last sub-package from *util* contains the creators of SmartModels redefinitions. Like for types and values, their hierarchy was designed to maximize the automatic control of the creation of redefinitions in the EMF generated editor. I used Java multiple inheritance for interfaces to express the common redefinition types between parameters and characteristics and simple polymorphism for the redefinition types which apply only to parameters or characteristics.

For example, there is *SmartModels_RedefinitionsFreeEditor* which means that the corresponding parameter can redefine its value to any value valid for its type. This redefinition editor can be an option to any type of redefinition and that is why it inherits from all concrete redefinition types (i.e., primitive types, collection type, enumeration type, atom type ...). On the other hand, there is *SmartModels_RedefinitionPairOfAtom* which can be used only for a characteristic of a DerivedAtom and that is why it will inherit only from *SmartModels_RedefinitionForCharacteristicValuesOfDerivedAtom Redefined* (and not from parameter values).

This hierarchy of redefinition creators is parallel to the built-in redefinition hierarchy and in the next phase of transforming the model (see section 5.3.3) I needed to map each one of them to the equivalent redefinition types from the Built-In plug-in. The *mapToBuiltInRedefinitionName* method is a cross-cut function for the editor, but it serves the mapping I need in the transformation phase and it returns the name of the related redefinition type.

This section presented the Meta-Data Editor entities that help us write SmartModels business-models. All of them are defined in the EMF model file called "*smartmodels_editor.ecore*". Based on this *ecore* model of the editor, the three levels of EMF code-generation explained in section 5.2.2 and the Built-In plug-in presented in section 5.3.1, EMF generates the "*.edit*" and the "*.editor*" plug-ins.

The entities of the Built-In plug-in cannot be edited by the programmer, but they are described using the Meta-Data Editor. That is why for the editor, the EMF generated the "*.editor*" plug-in which is also a Rich-Client Platform (RCP) application.

At this point there are two important issues I need to explain. First of all I need to remember that in order to be able to use the meta-object protocol (*mop* - see section 3.2) I need to initialize the SmartModels kernel. This is done in the *MetaDataEditorEditorAdvisor* before running the application.

Then I customized the wizard that guides a user through the creation of a new SmartModels model. In order to achieve this I modified the creation of the initial object in the class org.smartmodels.editor.meta_model. presentation.Meta_modelModelWizard.

By default, the EMF generated editor provides extension points for all non-empty packages in the ecore model. In SmartFactory there is no reason to describe some entities independent from a model. That is why I created an object called *SmartModelsEditor* which is the only valid container for a SmartModels model and I suppressed all the other extension points.

Next chapter presents an example of writing a business-model using SmartModels Meta-Data Editor. There are many references to this section and screen-shots to better understand and visualize the EMF RCP generated editor.

### 5.3.3. The Transformer Plug-In

SmartFactory is a software factory that aims to maximize the use of the tools provided by Eclipse framework. The Meta-Data Editor is generated by the EMF CodeGen.Editor plug-in and I also want to reuse this code-generator for my business-model. In this way I can regenerate and reuse all the evolving features that EMF will provide.

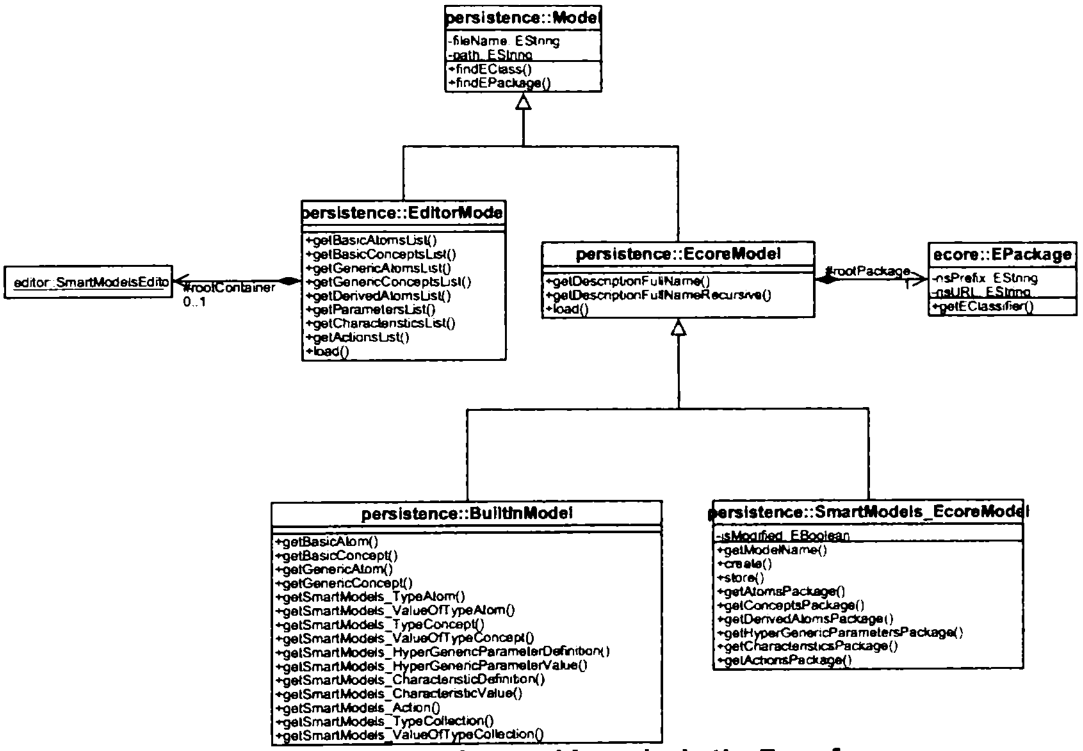

persistence::Model
-fileName EString
-path EString
+findEClass()
+findEPackage()
persistence::EditorModel
+getBasicAtomsList()
+getBasicConceptsList()
+getGenericAtomsList()
+getGenericConceptsList()
+getDerivedAtomsList()
+getParametersList()
+getCharacteristicsList()
+getActionsList()
+load()
editor::SmartModelsEditor
#rootContainer
0..1
persistence::EcoreModel
+getDescriptionFullName()
+getDescriptionFullNameRecursive()
+load()
#rootPackage
1
ecore::EPackage
-nsPrefix EString
-nsURL EString
+getEClassifier()
persistence::BuiltInModel
+getBasicAtom()
+getBasicConcept()
+getGenericAtom()
+getGenericConcept()
+getSmartModels_TypeAtom()
+getSmartModels_ValueOfTypeAtom()
+getSmartModels_TypeConcept()
+getSmartModels_ValueOfTypeConcept()
+getSmartModels_HyperGenericParameterDefinition()
+getSmartModels_HyperGenericParameterValue()
+getSmartModels_CharacteristicDefinition()
+getSmartModels_CharacteristicValue()
+getSmartModels_Action()
+getSmartModels_TypeCollection()
+getSmartModels_ValueOfTypeCollection()
persistence::SmartModels_EcoreModel
-isModified EBoolean
+getModelName()
+create()
+store()
+getAtomsPackage()
+getConceptsPackage()
+getDerivedAtomsPackage()
+getHyperGenericParametersPackage()
+getCharacteristicsPackage()
+getActionsPackage()

**Figure 34. Persistence hierarchy in the Transformer**

The problem is that the editor saves the resources in an XML encoding stream, but not *ecore* format, because SmartModels' entities add more information to the standard *ecore* entities. Therefore, in order to use the EMF CodeGen for generating code for SmartModels models they need to be transformed according to the ecore format.

This role is accomplished by the SmartModels_Transformer Plug-In (from now on I will call it "*the Transformer*") which can be found as a runtime library "*SmartModels_Transformer_PlugIn.jar*" in the SmartFactory framework. This plug-in makes a contribution to the menu bar which has the same name as the plug-in and adds the action (called *Transform*) that will do the job.

The following paragraphs will present the transformer architecture and will address implementation issues. Two practical examples of how it can be used will be given in chapter 6.

The meta-model of the Transformer has two parts:

- a set of six components, each one of them dealing with a part of the transformation;
- a hierarchy of classes which help the transformer to handle the different types of model serialization.

In order to run the Transformer a user has to specify two sources: the Built-In ecore model file (the kernel of SmartFactory - see section 5.3.1) and the Meta-Data Editor model file (the output of the editor - see section 5.3.2), and one target: the *ecore* resource where the model is stored after transformation.

To ease the utilization of the Transformer, if a user needs to run it as a standalone plug-in, I designed a *wizard* similar to the EMF editor where he specifies this three files before running the Transformer. The wizard is a dialog window which authorizes the starting of the Transformer if all three files exist. At the end of the transformation the user will also be notified with a message dialog if the process succeeded or not.

In the previous paragraphs we have seen the different types of models the Transformer handles. The classes which encapsulate them constitute the *persistence* package and their architecture in my plug-in is presented in Figure 34.

The *Model* abstract class is the root of the classes that handle the persistency. It contains only the commonalities between all the classes that contract a type of a model representation:

- *fileName* – the name of the file that saves the content of a model resource (XML encodings);
- *findEPackage* and *findEClass* – utility methods used for recursive search of a package or a class when the type (return type, super-type, etc.) of an entity is linked to another resource (it needs to load another resource in order to make the right association) or it needs to find the appropriate entity in the updated model after transformation (for example in the editor I created two entities: AtomDriverEditor and its super-class AtomPersonEditor. Now the transformer needs to find the transformed SmartModels AtomPerson in order to set it as super-class for the new AtomDriver).

In order to benefit from Eclipse EMF tools, equally, the starting point of my approach, the kernel of SmartModels, and the transformed business-models, are described using an *ecore* model. The transformer reifies it through *EcoreModel* abstract class which is the super-type of all *ecore* models managed by the transformer. It has:

- the *rootPackage* reference to an org.eclipse.emf.ecore. EPackage object which is the container of the *ecore* model;
- the *load* method which loads into memory the XML resource specified by the *fileName*. It can generate a java.io.IOException if the loading fails, otherwise the *rootPackage* field will reference the root EPackage of the *ecore* model. The corresponding *save* method is not defined at this level because the transformer can serialize just the SmartModels transformed model and it cannot change the Built-In model;
- other methods used by the code-generator to build the Java full name of an *org.eclipse.emf.ecore.EClass*. The EMF provides for its named elements the possibility to get the full name only using the EMF.CodeGen features, but the transformer needs to generate a part of the code before (see next paragraphs).

The *BuiltInModel* is a bridge between the transformer and the Built-In plug-in entities providing them while transforming the Meta-Data Editor model (each entity from the editor needs to be attached to the right entity from the Built-In). It defines many accessors to entities like Basic and Generic Atoms, Concepts, HyperGenericParameter and Characteristic Definitions, Values, Types and so on. They return the EClass objects representing these entities from the loaded Built-In *ecore* model. Therefore, any resource which store a SmartModels transformed model will depend on the Built-In model resource (the EMF sample editor automatically loads all the set of the resources from which it depends when loading a model).

The *SmartModels_EcoreModel* deals with the model that results from the transformation phase. This class brings three new things:

- it provides accessors to all packages that form a SmartModels model and which contain the different first-class entities (Atoms, Concepts, DerivedAtoms, HyperGenericParameters, Characteris-tics and Actions). They are used by the other recursive functions when searching for a particular entity;
- the *create* method which creates a new model. This method initializes the SmartModels model by creating the root package and providing it to the transformer initializer;
- the *store* method which saves the model into an XML resource. The Transformer can modify and write only SmartModels models and that is why it provides this feature only at this level;

The Meta-Data Editor model is dynamically loaded by the *EditorModel*. It is also a source of data for the Transformer and acts similar to the *BuiltInModel* - as an interface to the business-model written by the editor (identifying and implementing accessors to the SmartModels entities creators from the editor). It differs from an ecore file as the container of the model is an org.smartmodels.editor.SmartModelsEditor object, but beyond that the loading is the same because it is also an XML resource.

The actual Transformer is composed of a set of six components each one of them dealing with one of the SmartModels first-class entities. The following paragraphs will emphasize the main insights on how these components work together to do the work. The order of launching them is not important, but it does matter to run all of them on a model in order to have a valid transformation. That is why in the next version of the SmartFactory prototype I will better customize the use of the Transformer and I will add on top of it a "manager" which will organize the component roles (now this role is taken by the plug-in default action).

Let us first explore the common super-class of all components. It is an abstract class because it declares an abstract method called *transform* that has to be implemented by all the components. The rest of the data and the behaviour are useful instruments in the transformation process of every type of entity.

Each component invokes entities from both sources (the *BuiltInModel* and the *EditorModel*) and affects the target: the SmartModels model that results after transformation. That is why all those three references to the models are kept in their super-class. In addition, the Transformer interface provides to all components a database of constants used during code-generation. For example:

- there is a standard policy of naming the accessors and here I have defined the prefixes and the names I use;
- the standard attributes names for parameters and characteristics, for types and values, for redefinitions, etc.;
- package names of the SmartModels entities;
- the list of constants used to name the annotations the transformer adds to the entities in order to customize the code-generation phase (see next paragraphs and section 5.3.4).

**Table 5. The Transformer components**

| Target | Operations |
|---|---|
| Concept | - identifies the basic and generic concepts and builds the hierarchy of classes;<br>- adds accessors for actions, parameters and characteristics. |
| Hyper Generic Parameter | - identifies and sets the type and the default value;<br>- builds the list of the redefinition values;<br>- adds an accessor to get the concept container. |
| Characteristic | - idem to HyperGenericParameterTransformer component, but dealing with the corresponding characteristic types and values. |
| Action | - creates the "execute" method according to the editor defined operation setting the right parameters and return type;<br>- adds an accessor to get the concept container;<br>- adds a set of accessors to the hyper-generic parameters,<br>characteristics and other action parts that the target action uses. |
| Atom | - identifies the basic and generic atoms and builds the hierarchy of classes;<br>- adds an accessor to the corresponding attached concept. |
| Derived Atom | - identifies the targets' generic atoms and builds the hierarchy of classes;<br>- adds an accessor to the corresponding attached generic concept (it is the same concept attached to the target's generic atom);<br>- set the hyper-generic parameter and characteristic values;<br>- build the list of the possible redefinition values. |

Besides data, the common behavior of all components is encapsulated in their super-class. Here are a partial list of useful methods implemented at the top level:

- *initOutputModel* – it parses the editor model and dynamically builds the skeleton of the SmartModels model identifying all the first-class entities. As a result the model it is initialized and during transformation each component will fill it with data;
- *capName* (*unCapName*) – they are imported from the EMF.CodeGen plug-in in order to help the code-generators to conform to the Java code naming conventions;
- *addAccessMethodToAction* and the equivalent *addAccessMethodToCustomization* in order to equip the Concepts with accessors to their semantic information: actions and parameters. As I already mentioned, in SmartModels approach, all of them are first-class entities, so both ends of the relationship between the containing concept and the semantics (represented by the parameters,

characteristics, actions, etc.) will define a reference to the other. Also, each action provides accessors to its interesting parameters and other action parts that it uses;

- methods to apply the singleton pattern to the entities that reify the semantic information (see section 3.4.2). EMF.CodeGen does not provide the features for describing a singleton;
- methods to implement the *mop* (see section 3.2);
- methods to generate code for additional SmartModels information on types and values, redefinitions, etc.

Table 5 briefly presents the Transformer components: the entities they build and the main transforming actions they operate. All of them automatically add the *mop* information and, as it was mentioned above, they implement the singleton pattern for the meta-level entities when it is the case.

The last, but very important mention, is the way the Transformer deals with SmartModels features that the EMF tools do not provide (I already assumed that the ouput transformed model has to be an *ecore* model). The most significant limitation SmartFactory has to deal with when using EMF tools and Java programming language is that there is no support for the meta-level (the semantic information). SmartFactory makes the compromise to solve this by:

- using the Transformer plug-in to add *org.eclipse.emf.ecore.EAnnotation* in order to encapsulate the semantic information. In actual fact these comments encapsulate Java pure code generated by the Transformer (in this way I am prepared to change this approach for the day when EMF and Java will provide support for meta-level);
- using the code-generator plug-in to adapt the EMF.CodeGen in order to take into account the annotations made by the transformer while generating the code.

### 5.3.4. The Code-Generator Plug-In

This plug-in has the role of adding value to the EMF ecore code-generator in order to take into account the SmartModels specific annotations attached by the Transformer. It does not represent a phase in developing with SmartFactory (like writing a business-model with the Meta-Data Editor or transforming it), but it reuses the EMF.CodeGen to support particularities of my approach. As a result, the SmartModels code-generator (from now on I will call it "*SM.CodeGen*") updates the run-time library org.eclipse.emf.codegen. ecore distributed on Eclipse as "*codegen.ecore.jar*".

In the current version of the SmartFactory prototype, SM.CodeGen plug-in affects two of the JetEmitter templates (see section 5.2.2): Class.javajet and FactoryClass.javajet. Therefore this plug-in has another special property (Eclipse project specific property): it is converted to a JET project and besides the common Java builders it uses the JET builder to compile the templates. The JET builder generates the Java classes from JET Emitter templates which in turn will generate the Java code of the business-models for the corresponding EMF ecore entities.

More templates will be updated as SmartFactory is developing: the util.Switch class will be customized to support different types of visits on the model instances (in the previous attempt to develop SmartFactory using SmartTools platform it automatically generated different kinds of Visitors [87]). In order to clearly separate the entry points where SmartFactory updated the JET templates, I organized all of them as distinct emitter parts, each one of them in different files which can be included in the EMF templates when needed and updated separately.

**Table 6. EMF Java JET Emitters adapted by SmartFactory**

| **Name** | **EMF Java JET Emitter Target** | **Task** | **Annotations** |
|---|---|---|---|
| AddBody OfMethods | Class | EMF does not currently support encapsulation of method bodies in the ecore models. However, SmartFactory identifies two situations when I need to encapsulate them in order to execute the model: the *get* accessor to the single instance of the SmartModels entities that are singleton and the call to the *execute* method of the actions from their concept container. | *source*: body *typed key*: statement |
| AddEReference Information | Class | it contains information needed to initialize features of derived atoms (concept instantiation and possible redefinition values), hyper-generic parameters and characteristics (type information, default value and redefinition values). More information will be added in the future. | *source*: value, redefinition, typeInformation, conceptInstantiation *typed key*: statement |
| AddImports | Class | it parses the business-model packages and adds the required *import* statements in the generated classes according to the SmartModels specific code from the annotations. | *source*: organizingImports *typed key*: organizing ImportStatements |
| AddMOPIn SmartModels Objects | Class | it inserts the *mop* – static initializer (see section 3.2) for each SmartModels entity. It also offers support for writing comments. | *source*: staticInit *typed key*: mopComment, mop |
| AddSingleton | Class | it inserts the specific code of the Singleton pattern (it is also a static initializer) for the first-class entities that can have only one instance in a SmartModels application (concepts, hyper-generic parameters, characteristics and actions). | *source*: staticInit *typed key*: singletonComment, singleton |

| | | | |
|---|---|---|---|
| AddSingleton InFactory | Factory Class | it identifies the singleton entities from the model and changes the code of the standard "*create*" methods from the EMF Factory: it replaces the regular call of the related constructor with a call to get the single instance. | none |
| Constants | Class and Factory Class | it holds the database of all literals used to identify between annotations, standard SmartModels package names for a model, names of features (for example, *execute* for actions, *getSingle* for singletons, *getAttachedConcept* for entities that have a dependency on a concept, etc.). This is synchronized with the same database from the Transformer. | none |
| GetSmartModels Type | Class and Factory Class | it is based on the standard package names from a model it identifies the SmartModels entities. It is useful for recognizing the singletons, but more use-cases will be added in the future. | none |

Table 6 enumerates the SmartFactory updated entry points in the JET generator emitters, the target EMF Java JET Emitter that it can apply to, the task that it takes and the EAnnotation source and keys through which the Transformer identifies them.

After all these phases, the resulting code-generator will act exactly the same as the original EMF.CodeGen with regard to the Eclipse platform. A user may create the EMF *genmodel* and generate the code. But it is important to remember that using the SmartModels.CodeGen means that the obtained implementation has to be attached to the Built-In plug-in as it will reference the SmartModels entities (*mop*, basic and generic concepts/atoms, types and values, redefinitions, etc.).

## 5.4. Conclusion and Future Work

For a better understanding of the interest of SmartModels approach, it was important to give an overview of the implementation of the prototype, called SmartFactory. This work stands between the model approach and the AOP (Aspect Oriented Programming) [66] dedicated to DSLs (Domain Specific Languages), but in a broader context as it uses the concept of software factory.

Therefore, this chapter introduced the tool support (SmartFactory) for the approach presented in this thesis (SmartModels). SmartFactory prototype deals with important implementation issues and it represents an interpretation and validation of the approach.

On the short term perspective, SmartFactory needs some more work for integrating into the editor all the other auxiliary phases: transformation of the model and generation of the code. On the long term perspective, the prototype needs to keep being updated to support the future enhancements in the SmartModels approach.

Due to the purpose of separating the concerns when developing each plug-in of the SmartFactory prototype, they were all kept independent. When they will be ready for integration, the plan is to design a plug-in called Manager, which will control all the Transformer components and will automate both the transformation and code-generation based on the model customized by the editor. In this case the wizard of the Transformer will not be necessary anymore, because the goal is to provide a complete solution so the designer can generate code automatically from inside the Meta-Data Editor.

SmartFactory also aims to improve the techniques to write and implement models for the description of derived atoms and applications. Through the definition of those models which are dedicated to enrich the meta-model itself, it looks for improving the quality and the percentage of the source code automatically generated so a software company can gain competitive advantages like:

- preserving company investment (future legacy code);
- following rapid technology evolution;
- reacting faster to technology changes;
- improving productivity.

# 6. Model Implementation Example

This chapter takes the discussion on the SmartFactory prototype a step forward in order to illustrate, through an example, how to use the complete solution. It presents a relevant case-study to evaluate the methodology to create business models and the technique to generate applications afterwards. It also responds to several feed-backs received after the presentation of [10] to demonstrate the use and feasibility of this approach with examples.

Section 0 already introduced the example of photo cameras domain and explored it while developing the SmartModels approach (see section 3.4). Chapter 5 presents implementation issues of the new prototype. What remains is to see how SmartFactory prototype works step-by-step from the description of the model to code generation. The organization of this chapter aims also to continue the argumentation developed in chapter 4 for one of the most important applications of SmartModels principles towards building Software Product Lines (SPL) (see section 2.4).

A Software Product Line is a group of products that share a common, managed set of features (SmartModels approach calls them Atoms). The products satisfy the specific needs of a particular market or mission, and are developed from a common set of core assets in a prescribed way (Concepts). [31] [76].

Companies achieved increased quality and significant reductions in cost and time to market when they started to produce a set of related products (Atoms) in an automatic line. SPL calls together the analysis, design and implementation activities for a whole family of systems. SPL's Architecture, also called reference architecture, is a generic architecture from which the model of each product can be derived [119]. The role of software product line architecture is to describe the commonalities and the variabilities of the products contained in the Product Line (PL) and, as such, to provide a common overall structure. One of the main implementation issues that the SPL undertakes is to use *parameterization* (see sections 3.4 and 4.1): entities can be defined as generic assets with a set of parameters (SmartModels Concepts) and each product binds these parameters in a specific way (SmartModels DerivedAtoms).

The designer of a product line that makes photo cameras may identify the following entities and parts in the process of modelling this domain:

- memory devices to storage the image or video files;
- communication ports for connectivity to transfer data;
- support for different types of batteries;
- photo camera entity itself, etc.

Figure 17 visualises the EMF ecore model of the photo camera example which this chapter will examine. The UML notation format can be visualized in Figure 35.

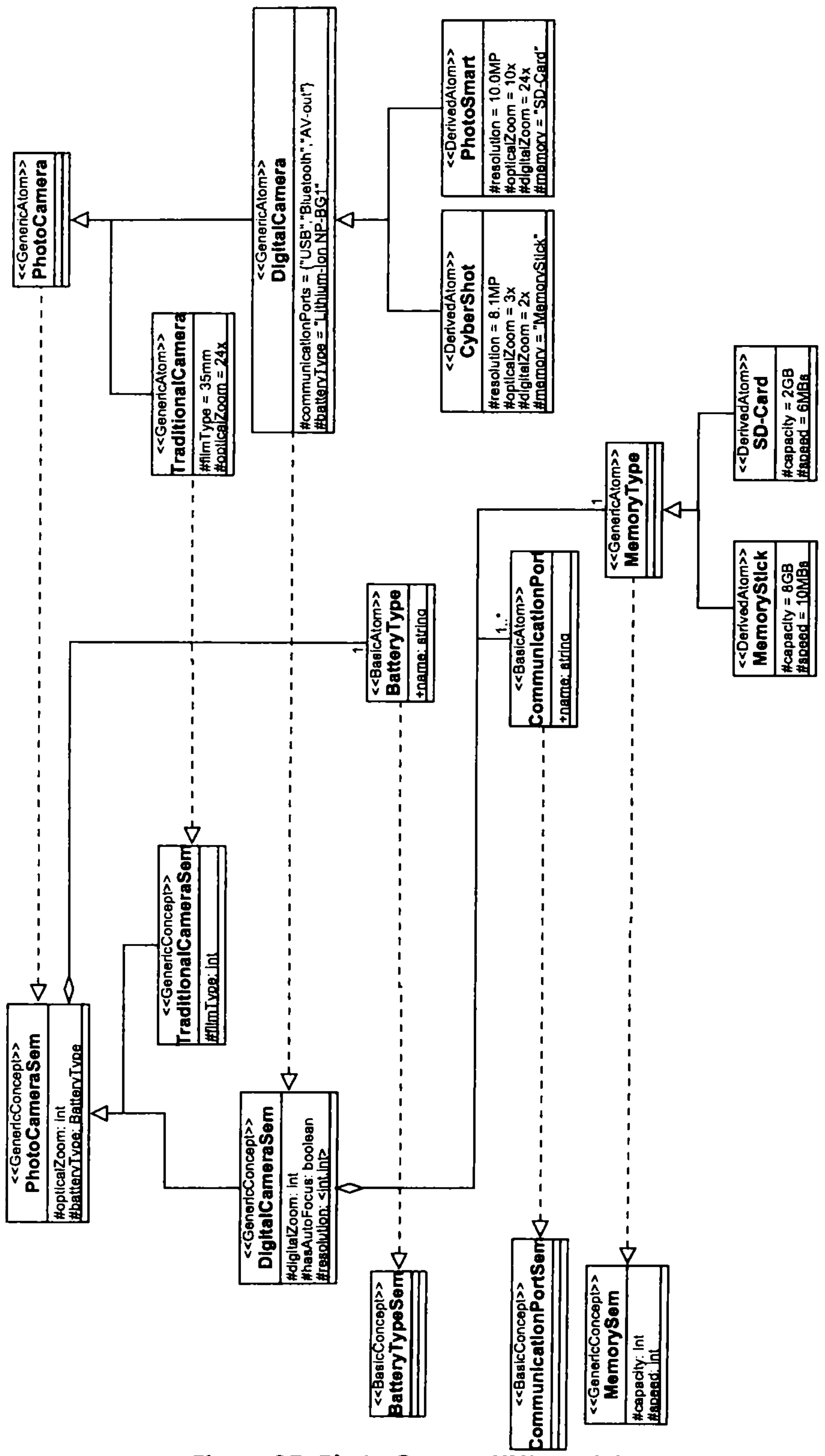


**Figure 35. Photo Camera UML model**

The methodology to build and use models in SmartFactory is made up of four steps:

- writing the business-model using the Meta-Data Editor (see section 5.3.2);
- transforming the model using the Transformer (see section 5.3.3) to an EMF ecore compliant model (making use of annotations);
- generating the code using the EMF-CodeGen updated generator (see section 5.3.4) by interpreting the annotation made by the Transformer;
- building applications implementing the Switch - a visitor on the model automatically generated by EMF.CodeGen in the previous step.

Next sections address each one of these steps and investigate their methodology similar to the creation of a tutorial for SmartModels designers. Each section is decorated with screen-shots from the various plug-ins and tools used to develop on Eclipse platform and also editor diagrams in order to better understand and visualize the use of each component of SmartFactory.

## 6.1. Writing the Model

This section explains, step-by-step, the procedures to write a business model using the editor provided by the SmartFactory prototype. It assumes that the platform has already installed the kernel of the SmartModels (the built-in kernel presented in section 5.3.1).

The SmartModels Meta-Data Editor is a rich-client platform application and therefore it can be used either as an Eclipse plug-in or as a standalone Eclipse application. Using Eclipse to launch an RCP application, it is basically the same as launching an Eclipse Application [36]. However, there are some extra steps required to specify that the designer wants to use the generated application instead of the default one (`org.eclipse.ui.ide.workbench`) and to make sure that only the needed plug-ins are loaded at runtime.

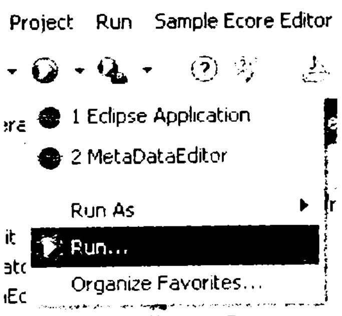
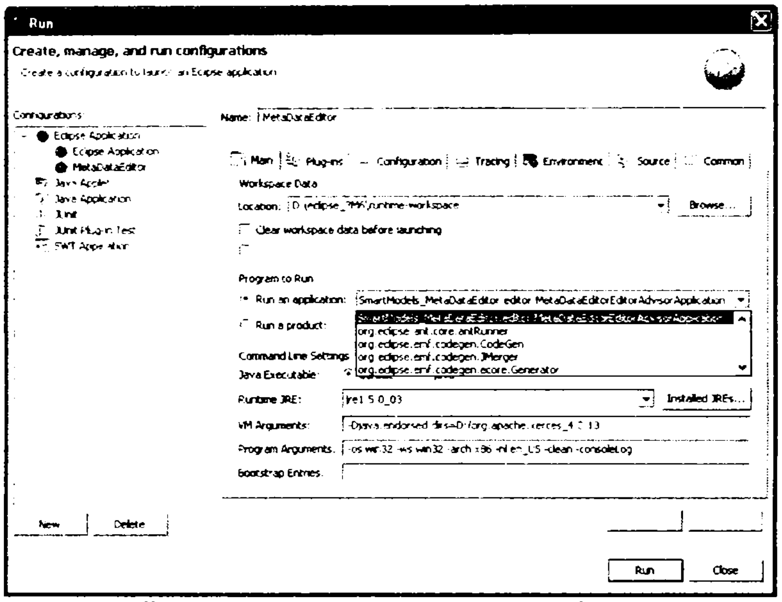


**Figure 36. Eclipse *Run* menu**

Before being able to actually write the model entities, there is a set of phases to go through to create an Eclipse running configuration and to know how to use the editor's wizard which helps to select the model's root entity. The steps below will guide through the process of creating the Eclipse Run configuration [35]:

1. In the Java perspective click the *Run* menu (see Figure 36):
2. Create a new *Eclipse Application* configuration (Figure 37 shows some of the tailored settings of an RCP):



Run

Create, manage, and run configurations

Create a configuration to launch an Eclipse application

Configurations:
- Eclipse Application
  - Eclipse Application
  - MetaDataEditor
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- SWT Application

Name: MetaDataEditor

Main | Plug-ins | Configuration | Tracing | Environment | Source | Common

Workspace Data

Location: D:\eclipse_3M6\runtime-workspace

Clear workspace data before launching

Program to Run

Run an application: SmartModels_MetaDataEditor.editor.MetaDataEditorEditorAdvisorApplication

Run a product:

- SmartModels_MetaDataEditor.editor.MetaDataEditorEditorAdvisorApplication
- org.eclipse.ant.core.antRunner
- org.eclipse.emf.codegen.CodeGen
- org.eclipse.emf.codegen.JMerger
- org.eclipse.emf.codegen.ecore.Generator

Command Line Settings

Java Executable:

Runtime JRE: jre1.5.0_03

VM Arguments: -Djava.endorsed.dirs=D:\org.apache.xerces_4.0.13

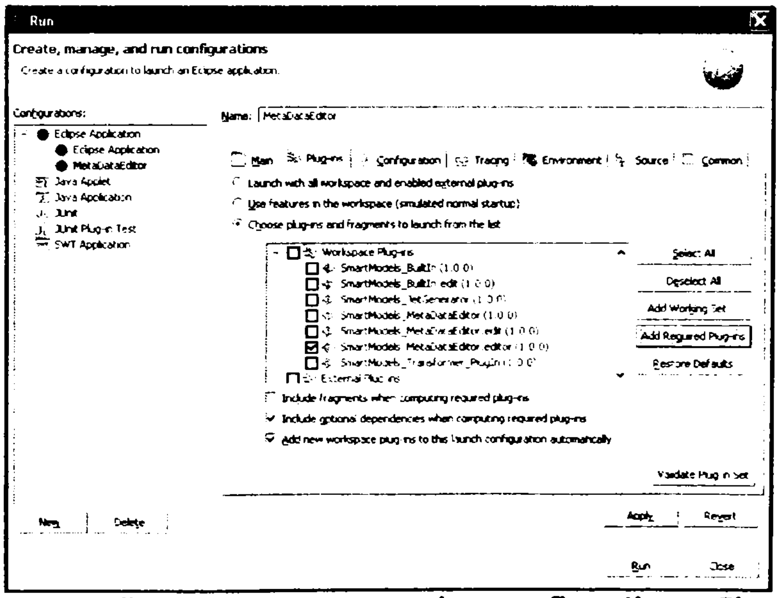
Program Arguments: -os win32 -ws win32 -arch x86 -nl en_US -clean -consoleLog

Bootstrap Entries:

New | Delete | Run | Close

**Figure 37. Eclipse: create, manage and run configurations – Main**

3. Specify the plug-ins that need to be available when running the application (see Figure 38). If the editor is executed as an Eclipse RCP application than there is no reason at run-time to load all the available plug-ins – it is time and memory consuming. Therefore, this step is just to ensure this optimization and it is important due to the fact that, when Eclipse loads the required plug-ins to run `SmartModels_MetaDataEditor.editor.MetaDataEditorEditorAdvisor`, it automatically loads all the optional dependencies declared by all the used plug-ins).



Run

Create, manage, and run configurations

Create a configuration to launch an Eclipse application.

Configurations:
- Eclipse Application
  - Eclipse Application
  - MetaDataEditor
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- SWT Application

Name: MetaDataEditor

Main | Plug-ins | Configuration | Tracing | Environment | Source | Common

Launch with all workspace and enabled external plug-ins

Use features in the workspace (simulated normal startup)

[✓ Choose plug-ins and fragments to launch from the list]

- Workspace Plug-ins
  - SmartModels_BuiltIn (1.0.0)
  - SmartModels_BuiltIn.edit (1.0.0)
  - SmartModels_JetGenerator (1.0.0)
  - SmartModels_MetaDataEditor (1.0.0)
  - SmartModels_MetaDataEditor.edit (1.0.0)
  - [✓ SmartModels_MetaDataEditor.editor (1.0.0)]
  - SmartModels_Transformer_PlugIn (1.0.0)
- External Plug-ins

Include fragments when computing required plug-ins

[✓ Include optional dependencies when computing required plug-ins]

[✓ Add new workspace plug-ins to this launch configuration automatically]

Select All | Deselect All | Add Working Set | Add Required Plug-ins | Restore Defaults | Validate Plug-in Set

New | Delete | Apply | Revert | Run | Close

**Figure 38. Eclipse: create, manage and run configurations – Plug-ins**

4. Click *Run* and the editor was just launched (see Figure 39).

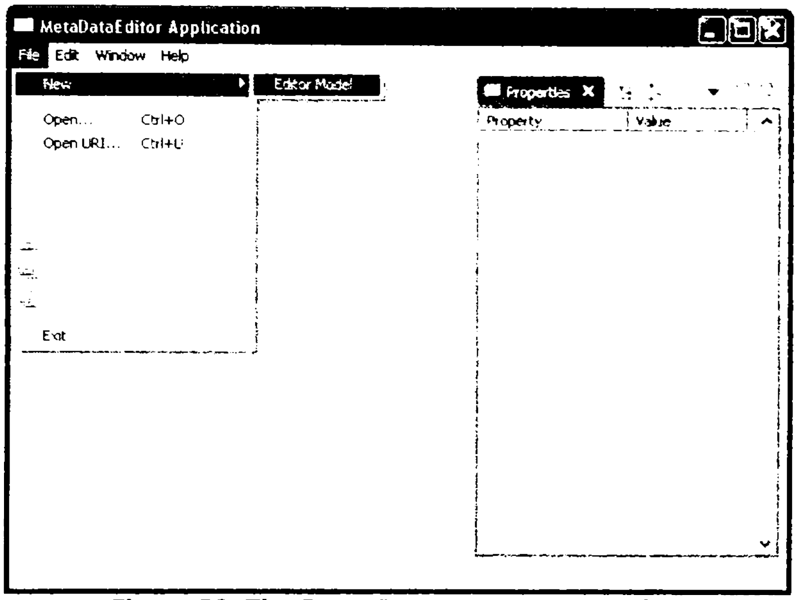


**Figure 39. The SmartFactory Meta-Data Editor**

The Meta-Data Editor provides a user-friendly wizard to start creating a model. It requires the user to specify:

- the file system location (interpreted in the Eclipse context as a resource) where the new model will be serialized. If the file extension is not mentioned, the application will not be able to locate the appropriate editor in order to manipulate the objects and it will display an error message;
- the meta-model object: the root entity of the model which is going to be described. EMF.Editor allows by default to create a model having as root any non-abstract class of any package from the meta-model. Therefore, in general use case, a designer may choose to work only on a part of the model at a moment, but the Meta-Data Editor deliberately removed all the other extensions and let only `org.smartmodels. editor.SmartModelsEditor` objects to be created (see section 5.3.2);
- the XML encoding used to serialize the model representation.

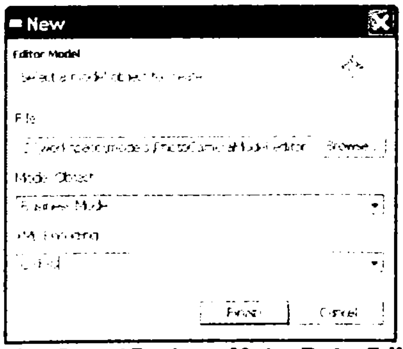


**Figure 40. The SmartFactory Meta-Data Editor Wizard**

Now the designer arrived at the phase where he can actually start writing the model. Figure 35 presents the UML Class Diagram of photo cameras product line. Next pages print five screen-shots of the Meta-Data Editor's main menus which help modelling in SmartFactory:

- Figure 41 shows how to add generic concepts in a model;
- Figure 42 shows how to enrich a concept with actions;
- Figure 43 illustrates how to customize a generic atom by setting its properties like: the name, the attached concept which stores its meta-information, the parent from the atom's hierarchy, etc.;
- there are two ways to derive atoms (products): by concept instantiation – setting values for parameters and characteristics of its the meta-information (see Figure 44), or by redefinition of a part or all values of another derived atom parent (see Figure 45).

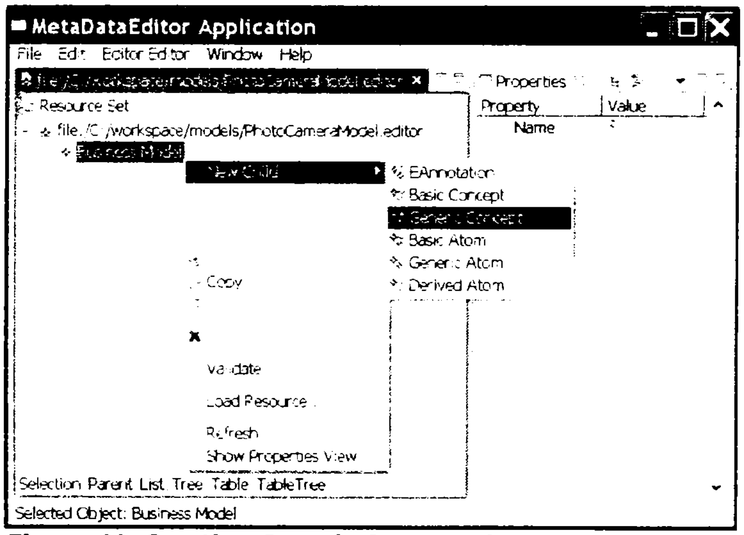


Figure 41. Creating Generic Concepts in Meta-Data Editor

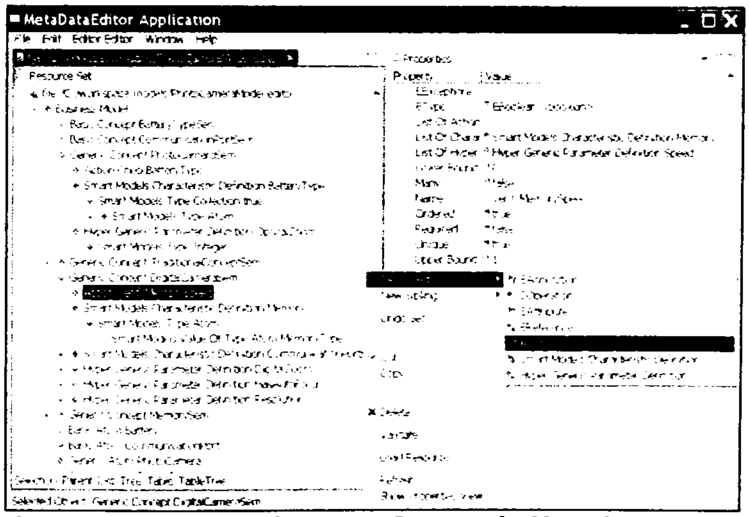
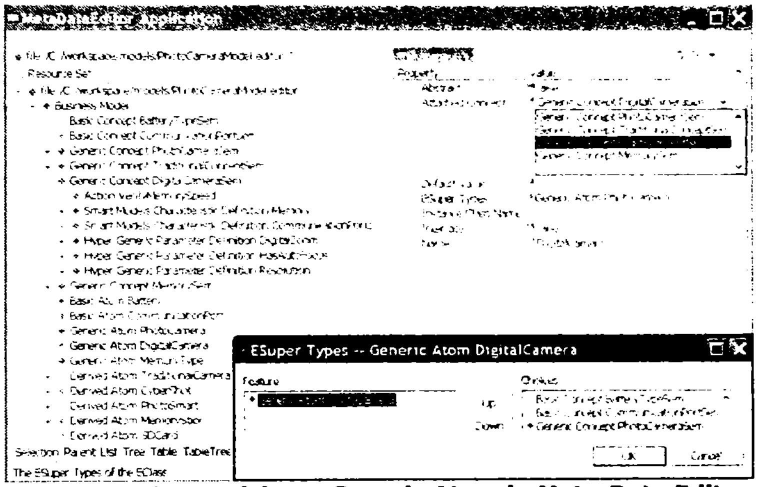


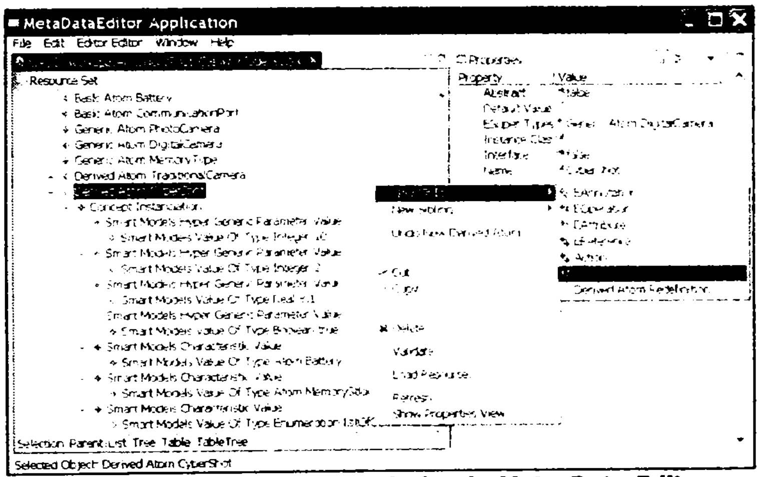
Figure 42. Adding Actions to a Concept in Meta-Data Editor


MetaDataEditor Application
ESuper Types -- Generic Atom DigitalCamera
Feature
Choices
Up
Down
OK
Cancel

Resource Set
Business Model
Basic Concept BatteryTypeSem
Basic Concept CommunicationPortSem
Generic Concept PhotoCameraSem
Generic Concept DigitalCameraSem
Hyper Generic Parameter Definition DigitalZoom
Hyper Generic Parameter Definition HasAutoFocus
Hyper Generic Parameter Definition Resolution
Basic Atom Battery
Generic Atom PhotoCamera
Generic Atom DigitalCamera
Derived Atom TraditionalCamera
Derived Atom CyberShot
Derived Atom PhotoSmart
Derived Atom SDCard
Selection Parent List Tree Table TableTree
The ESuper Types of the EClass

Property | Value
Abstract | false
ESuper Types | Generic Atom PhotoCamera
Generic Concept PhotoCameraSem

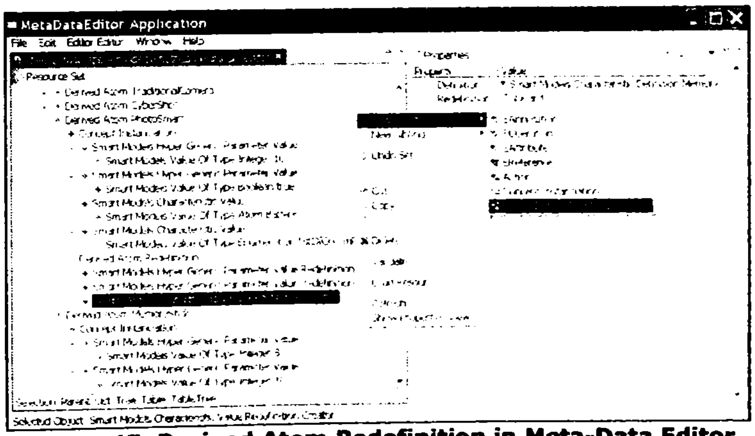
Figure 43. Customizing a Generic Atom in Meta-Data Editor



Resource Set
Basic Atom Battery
Basic Atom CommunicationPort
Generic Atom PhotoCamera
Generic Atom DigitalCamera
Generic Atom MemoryType
Derived Atom TraditionalCamera
Concept Instantiation
Smart Models Hyper Generic Parameter Value
Smart Models Value Of Type Integer 10
Smart Models Hyper Generic Parameter Value
Smart Models Value Of Type Integer 2
Smart Models Hyper Generic Parameter Value
Smart Models Value Of Type Real 4.1
Smart Models Hyper Generic Parameter Value
Smart Models Value Of Type Boolean true
Smart Models Characteristic Value
Smart Models Value Of Type Atom Battery
Smart Models Characteristic Value
Smart Models Value Of Type Atom MemoryStick
Smart Models Characteristic Value
Smart Models Value Of Type Enumeration 1280
Selection Parent List Tree Table TableTree
Selected Object: Derived Atom CyberShot

Property | Value
Abstract | false
Default Value |
ESuper Types | Generic Atom DigitalCamera
Instance Class |
Interface | false
Name | CyberShot

Figure 44. Concept Instantiation in Meta-Data Editor



Resource Set
Derived Atom TraditionalCamera
Derived Atom CyberShot
Derived Atom PhotoSmart
Concept Instantiation
Smart Models Hyper Generic Parameter Value
Smart Models Value Of Type Integer 10
Smart Models Characteristic Value
Derived Atom Redefinition
Smart Models Hyper Generic Parameter Value Redefinition
Derived Atom MemoryStick
Concept Instantiation
Selection Parent List Tree Table TableTree
Selected Object: Smart Models Characteristic Value Redefinition

Figure 45. Derived Atom Redefinition in Meta-Data Editor

The following paragraphs will briefly present some of the EMF editor's like properties of an entity which have had a decisive influence on the way the editor customizes the manipulation of SmartModels objects.

An `org.eclipse.emf.ecore.EReference` is *containment* if it represents by-value content, otherwise it can reference an object which belongs to another container (therefore, it can be even from another model and as a result from another resource). If *containment* is set to `true` then the EMF.Edit plug-in provides the possibility to create the corresponding object when creating the container ("*Create Child*" action in the context menu). If *containment* is set to `false` then it will fill up a drop-down list (if the reference has multiplicity of value *one* [1]) or a list box (if multiplicity is of value *many* [*]) with all the instances of that object in the model to reference them from the container.

For example, a hyper-generic parameter belongs to a GenericConcept and it cannot be defined independent from its attached concept. In opposition to it, there is a hyper-generic parameter value which does not contain its type (multiplicity: [1]), but it just references it. That is why values have *containment* property set to `false` when pointing out there type definition. In the same way the super-type of an Atom is not created every time a user extends its hierarchy or derives new atoms.

An `org.eclipse.emf.ecore.EStructuralFeature` is *volatile* if it represents a computed and not a data field. For such a feature the EMF.CodeGen will not generate an attribute, but accessors (the *get* method is automatically generated, while the *set* method is generated only if the *changeable* property is set to `true`). For example, the attribute *numberOfElements* of the SmartModels_TypeTuple is automatically computed based on the number of its SmartModels_TupleParts.

## 6.2. Transforming the Model

The Meta-Data Editor provides the needed functionality to write model entities compliant to the SmartModels approach. However, this is an extension of EMF plug-in to support the creation of SmartModels key elements. In order to benefit from the EMF standard and its tools for generating source-code, this model needs to be transformed into an EMF Ecore compliant model. The added functionality will be customized automatically through EAnnotations (see section 5.3.2) which will be later interpreted by the extension of the EMF code-generator (see section 6.3).

This is the motivation for having to go through a transformation phase and this plug-in is called the "Transformer". In order to run it, a user has to invoke the "*Transform*" action form the SmartModels_Transformer menu (see Figure 46).

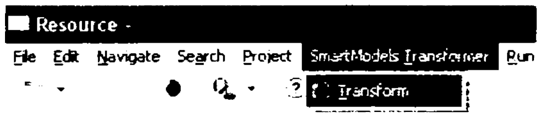
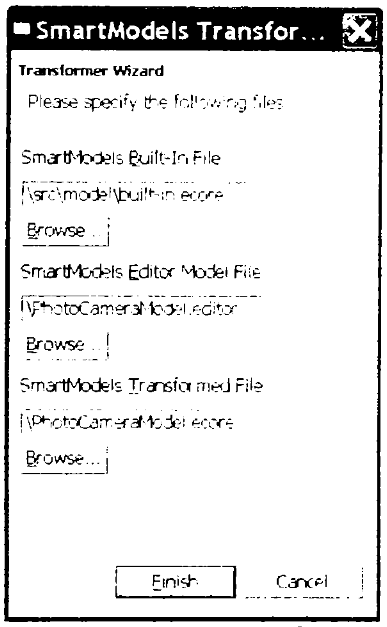

Resource -
File Edit Navigate Search Project SmartModels Transformer Run
Transform

**Figure 46. SmartFactory Transformer Plug-In**

To ease to work of the designer, the Transformer comes with a customized wizard (see Figure 47) which helps to specify the target three models that the Transformer needs to do its job:

- the *ecore* model of the SmartModels_BuiltIn plug-in which offer support for all SmartModels entities (`SmartModels_BuiltIn/src/model/built-in.ecore`);
- the designer business-model created with the SmartModels Meta-Data Editor in the previous phase (`SmartModels_MetaDataEditor/src/models/PhotoCameraModel.editor`);
- the new *ecore* resource where the Transformer will save the EMF Ecore compliant representation of the model (`SmartModels_JetGenerator/src/models/PhotoCameraModel.ecore`).

These parameters are handled by the `smartModels_Transformer.actions.Transform` action.


SmartModels Transfor...

Transformer Wizard

Please specify the following files

SmartModels Built-In File

\src\model\built-in.ecore

SmartModels Editor Model File

\PhotoCameraModel.editor

SmartModels Transformed File

\PhotoCameraModel.ecore

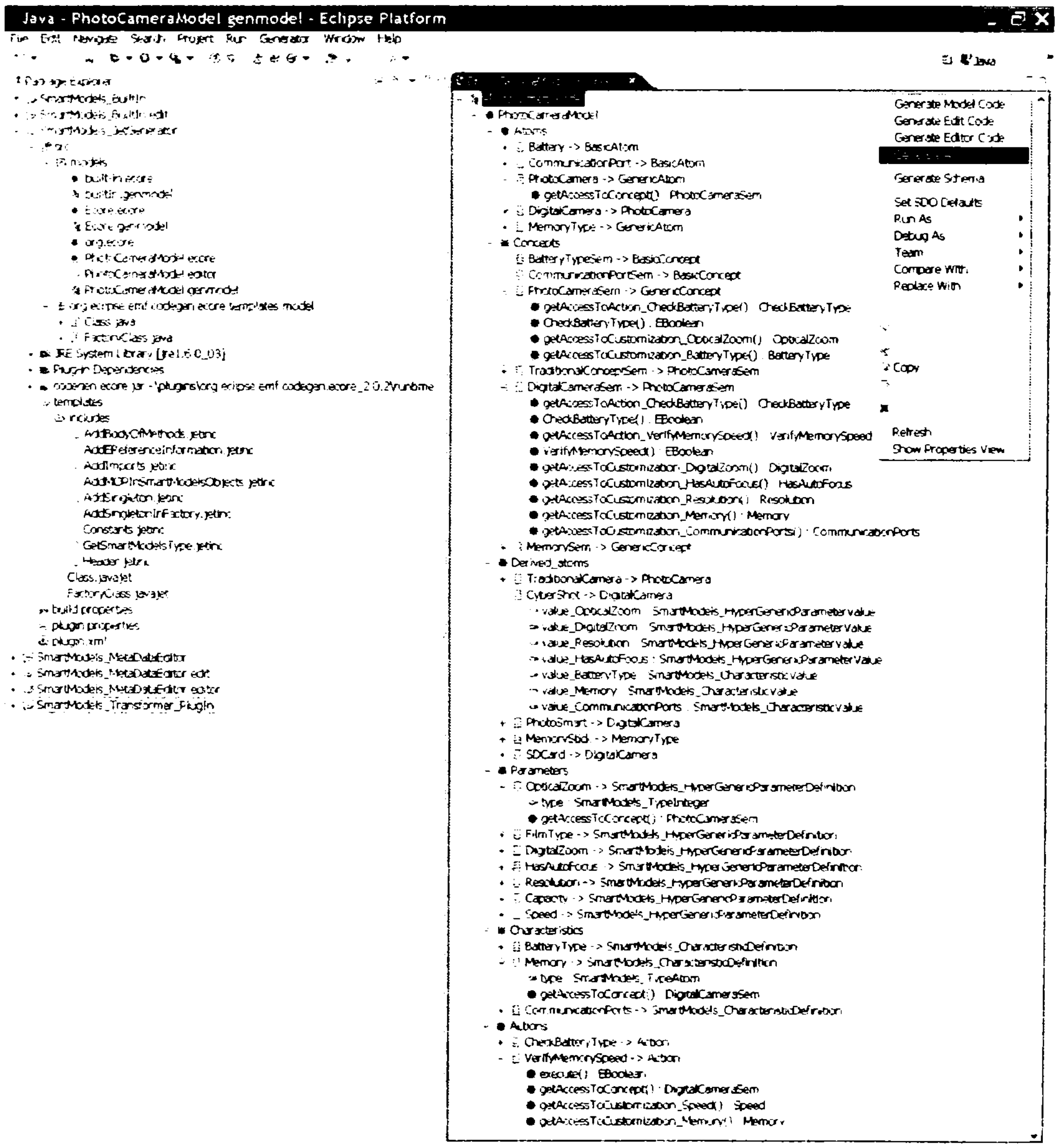
**Figure 47. SmartFactory Transformer Wizard**

## 6.3. Generating the code

Section 5.3.4 already mentioned that SmartFactory code-generator reuses the Eclipse EMF code generator based on Ecore models. The transformation phase described in the previous section explained how a model, written with in SmartFactory Meta-Data Editor (which provides all the functionality to benefit from the richness of the meta-model entities) now it is converted into an EMF Ecore compliant model.

Therefore, the SmartFactory code-generator is an extension of the Eclipse EMF model to code-generator which adds the methods to interpret the specific meta-information encapsulated prior to this phase in the form of entity annotations. That is why, an important precondition in order to be able to use the SmartFactory code-generator in such a way that it takes into account the meta-information, is mandatory to replace the default Eclipse code-generator plug-in (`org.eclipse.emf.codegen.ecore`) from the run-time library *"codegen.ecore.jar"*, with the one updated and provided by the SmartFactory prototype.

Afterwards, the code-generation process follows the same patterns like for any standard Eclipse EMF Ecore model resource.



```
Package Explorer
+ SmartModels_BuiltIn
+ SmartModels_BuiltIn.edit
- SmartModels_JetGenerator
  - src
    - models
        built-in.ecore
        builtin.genmodel
        Ecore.ecore
        Ecore.genmodel
        org.ecore
        PhotoCameraModel.ecore
        PhotoCameraModel.editor
        PhotoCameraModel.genmodel
    - org.eclipse.emf.codegen.ecore.templates.model
      + Class.java
      + FactoryClass.java
  + JRE System Library [jre1.6.0_03]
  + Plug-in Dependencies
  + codegen.ecore.jar - \plugins\org.eclipse.emf.codegen.ecore_2.0.2\runtime
    templates
      includes
        AddBodyOfMethods.jetinc
        AddEReferenceInformation.jetinc
        AddImports.jetinc
        AddMOPInSmartModelsObjects.jetinc
        AddSingleton.jetinc
        AddSingletonInFactory.jetinc
        Constants.jetinc
        GetSmartModelsType.jetinc
        Header.jetinc
      Class.javajet
      FactoryClass.javajet
    build.properties
    plugin.properties
    plugin.xml
+ SmartModels_MetaDataEditor
+ SmartModels_MetaDataEditor.edit
+ SmartModels_MetaDataEditor.editor
+ SmartModels_Transformer_PlugIn
```

```
- PhotoCameraModel
  - Atoms
    + Battery -> BasicAtom
    + CommunicationPort -> BasicAtom
    - PhotoCamera -> GenericAtom
        getAccessToConcept() PhotoCameraSem
    + DigitalCamera -> PhotoCamera
    + MemoryType -> GenericAtom
  - Concepts
      BatteryTypeSem -> BasicConcept
      CommunicationPortSem -> BasicConcept
    - PhotoCameraSem -> GenericConcept
        getAccessToAction_CheckBatteryType() CheckBatteryType
        CheckBatteryType() . EBoolean
        getAccessToCustomization_OpticalZoom() OpticalZoom
        getAccessToCustomization_BatteryType() . BatteryType
    + TraditionalConceptSem -> PhotoCameraSem
    - DigitalCameraSem -> PhotoCameraSem
        getAccessToAction_CheckBatteryType() CheckBatteryType
        CheckBatteryType() . EBoolean
        getAccessToAction_VerifyMemorySpeed() VerifyMemorySpeed
        VerifyMemorySpeed() EBoolean
        getAccessToCustomization_DigitalZoom() DigitalZoom
        getAccessToCustomization_HasAutoFocus() HasAutoFocus
        getAccessToCustomization_Resolution() Resolution
        getAccessToCustomization_Memory() : Memory
        getAccessToCustomization_CommunicationPorts() : CommunicationPorts
    + MemorySem -> GenericConcept
  - Derived_atoms
    + TraditionalCamera -> PhotoCamera
      CyberShot -> DigitalCamera
        value_OpticalZoom SmartModels_HyperGenericParameterValue
        value_DigitalZoom SmartModels_HyperGenericParameterValue
        value_Resolution SmartModels_HyperGenericParameterValue
        value_HasAutoFocus : SmartModels_HyperGenericParameterValue
        value_BatteryType SmartModels_CharacteristicValue
        value_Memory SmartModels_CharacteristicValue
        value_CommunicationPorts . SmartModels_CharacteristicValue
    + PhotoSmart -> DigitalCamera
    + MemoryStick -> MemoryType
    + SDCard -> DigitalCamera
  - Parameters
    - OpticalZoom -> SmartModels_HyperGenericParameterDefinition
        type : SmartModels_TypeInteger
        getAccessToConcept() : PhotoCameraSem
    + FilmType -> SmartModels_HyperGenericParameterDefinition
    + DigitalZoom -> SmartModels_HyperGenericParameterDefinition
    + HasAutoFocus -> SmartModels_HyperGenericParameterDefinition
    + Resolution -> SmartModels_HyperGenericParameterDefinition
    + Capacity -> SmartModels_HyperGenericParameterDefinition
    + Speed -> SmartModels_HyperGenericParameterDefinition
  - Characteristics
    + BatteryType -> SmartModels_CharacteristicDefinition
    - Memory -> SmartModels_CharacteristicDefinition
        type SmartModels_TypeAtom
        getAccessToConcept() DigitalCameraSem
    + CommunicationPorts -> SmartModels_CharacteristicDefinition
  - Actions
    + CheckBatteryType -> Action
    - VerifyMemorySpeed -> Action
        execute() EBoolean
        getAccessToConcept() : DigitalCameraSem
        getAccessToCustomization_Speed() Speed
        getAccessToCustomization_Memory() Memory
```

**Figure 48. SmartFactory Code Generator**

## 6.4. Conclusion

This chapter presented the use of tool support for the SmartModels approach. SmartFactory prototype deals with important implementation issues and this chapter exemplified its utilization. It explores a relevant case-study from the domain of lines of production of photo cameras, in order to evaluate the methodology to create models and the technique to generate applications afterwards.

It is a step-by-step description on how to write a model in the prototype's meta-editor, how to transform it to an Eclipse EMF Ecore compliant model (making use of annotations) in order to reuse the EMF code generator and then how all of this is setting the stage for building the modelled domain applications.

The purpose of this chapter was to demonstrate, through a real case modelling scenario, the feasibility of the approach and how easy it can be exploited once the know-how of the domain is encapsulated in a SmartModels model.

# 7. Conclusions and Perspectives

The world of software engineering arrived to a paradigm shift from *object technology* to *model technology*, from *object composition* to *model transformation*. From objects and components, we can see other evolving trends like: processes, rules, services. Therefore, "*model once, generate everywhere*" is the assertion that put a mark on the paradigm shift generated by the MDA, but the road to model engineering takes time.

In the past two decades, there were many changes in computer science that had an influence upon the way an application must be developed:

- The emergence of Internet which implied applications no longer stand-alone, but rather distributed. Therefore, from now on data communication between applications and users must be taken into account during the whole application life-cycle. One important point is to choose a well adapted data exchange format.
- The proliferation of new component technologies. It is difficult to choose the right and most promising. For instance, to obtain a component based application, a developer must choose between, at least, three component technologies: CCM (CORBA Component Model), EJB (Enterprise Java Bean), or Web Services.
- The widespread of computer science. Users may have now different knowledge levels, different needs, a wide range of visualization devices, and specific activity domains.
- The market pressure: in order to be efficient and competitive a company must quickly and cheaply adapt its software to new user needs and technologies. Time to market a software product is smaller and companies face a huge pressure on this aspect, but this factor should not shortcircuit the absolute need for quality.

To cope with these changes, applications need to be more open, adaptable and ready for fast evolution. The object-oriented approach does not provide all the solutions even if it represents a valuable basis for the description of further approaches. This remark can also be applied to component-based software engineering paradigm. In particular, they do not provide a correct answer to the continuous evolution of the technologies: keeping applications up-to-date according to the evolution of technologies is too much time-consuming.

This thesis promotes the idea that the most promising way to address software engineering is *to provide an approach centred on models which captures the know-how of a domain, independently from both the software platform and the possible applications*.

## 7.1. Contributions

The state of software engineering art (see chapter 0) drew the map of a whole set of new programming paradigms and emergent ideas into MDA approach such as: Domain Driven Development (DDD) [27], Unified Modelling Language (UML) [80], Aspect Oriented Programming (AOP) [66], Subject Oriented Programming (SOP) [48], Component Programming [100] [49] and generative programming [26]. Therefore, now it is no more important to develop only one software applied to a particular problem, but rather to design and develop a line (or a family) of software (SPL) [76] which takes into account features that may vary and makes it possible to minimize the costs and time of realization.

In this context, this thesis concentrates on providing a solution to respond to two main questions:

- *How to develop model-oriented software* (see chapter 3)*?* In other words, it presents SmartModels approach which offers a flexible and easy way to describe the know-how of a business domain into models independent from the technology. Technology evolves continuously in all aspects - platforms, programming languages, etc., and as a result it is important for companies to find a solution to avoid investing in placing their know-how directly at the level of a software platform, and to invest to a meta-level instead (using parameterized genericity).
- *How to build software product lines* (see chapter 4)*?* In other words, SmartModels provides a solution for describing concepts - a new method of abstracting the domain entities into the model level, methods to handle the commonalities, but also the differences between products, which in addition call for the need to impose constraints. As a result a company gains in terms of productivity and adaptability. It also fosters new opportunities for easier evolution of the semantics of the domain entities and for better reusability which again results in increasing companies' productivity.

Therefore, the thesis aims to take part in the modelling and the use of software product lines and the following sections identify its main contributions.

### 7.1.1. Defining an approach – *SmartModels*

SmartModels is a technology neutral approach which introduces a new meta-level on top of the classical programming entities. It deals with meta-modelling issues: besides OOP inheritance concept, the integration of genericity concept is an important ingredient for reuse and handling product variability. It intends to enrich the MDA emerging approaches in order to make easier the development of domain specific applications. This approach is original and may be distinguished from other approaches by the following characteristics:

- it introduces on top of the entities which structure the model (reification level - see section 3.4.3), a semantic layer which enables to define and factorize the basic functionalities related to the domain (see section 3.4.2);
- it provides a set of facilities, which relies on the two levels of the model (data and semantic models), in order to quickly build applications related to the model (see section 3.4.4);

- it ensures a clear separation between the model and the technologies which make the model executable by a software platform (see section 4.1).

### 7.1.2. Developing a solution based on models for product derivation in SPL

SmartModels proposes a technique (see chapter 4) for the integration of static and dynamic concerns in the models and clear entry points to insert constraints for coherence. Product variabilities are handled through:

- a clear differentiation between the semantic information and reification of the families of entities of a domain – SmartModels unambiguously separates the description of the structural features of each entity of a model from its semantics (see section 0);
- fundamental object technology well-known mechanisms like polymorphism and abstraction (see section 4.1.2);
- flexible management of genericity through concepts which encapsulate the meta-information of an entity – it is important to survey the role of the hyper-generic parameters to describe the semantics (see section 4.1.3);
- a good control over the dynamic aspects and constraints which insures model's consistency and which also guarantees the control of the semantics over the behaviour of the derived products – SmartModels actions (see section 4.1.4);
- deriving atoms in order to enrich the model and capture in the modelling phase as much as possible the commonalities and variabilities of the domain entities (see section 4.1.5).

### 7.1.3. Providing tool support – *SmartFactory*

The purpose of the tool support is to experiment and validate the approach: SmartFactory prototype deals with important implementation issues and represents an interpretation of SmartModels approach. It reduces the gap between modelling methodologies and programming languages, between the system's architects and the programmers. This work stands between the model approach and the AOP (Aspect Oriented Programming) [66] dedicated to DSLs (Domain Specific Languages), but in a broader context as it uses the concept of software factory. It provides a Meta-Data Editor, a mechanism for model transformation to be compliant with Eclipse EMF standard [36] and extends the EMF code-generator to support SmartModels approach features (see chapter 5).

All of the contributions mentioned above are accompanied by concrete examples for better assessment of SmartModels approach (see chapter 3 and chapter 4) and SmartFactory prototype (see chapter 6). They aim to clearly exemplify the modelling methodologies proposed by the approach which harmonize with the MDA principles. However, a new approach for the development of software based on models must ensure that software engineering skills are covered and improved in comparison with object-oriented and aspect-oriented approaches.

## 7.2. Limitations

Trying to predict the future achievements of modelling approaches in software engineering, as others have discovered, is risky. It is clear that the methodologies and technologies are changing so fast that SmartModels must continue to develop in order to keep up with the new trends and paradigms that emerge. Therefore, this section presents some limitations of SmartModels from the viewpoint of the current state both of the modelling art in general and the solution proposed by this approach:

- The niche, to which this approach applies, is the object-oriented software modelling and there is no effort spent or solution explored for the legacy software where the know-how is not from the world of object-oriented;
- The code generator relies on the efforts of Eclipse platform and therefore the adaptation of the implementation according to the execution contexts is bound to these approaches. They gather an important number of research communities and record an important progress right now, but still do not have support for all abstract entities of my approach. For this reason, the SmartModels' entities need customization in order to express at the implementation level the SmartModels' semantics and current solutions are not fairly straightforward (i.e., see section 6.2 on Class JetEmitter);
- There is a slightly more emphasis on designer effectiveness, productivity and following the technology evolution than correctness and quality. This is an important issue as the percent of the code automatically generated is still small and therefore, an application continues to require the input of the programmers. This means that the potential of programming errors is still high and testing phase still needs a long time. It also assumes that coding just some functionality in specific entry points in the code already generated automatically, may be harder to implement in isolation (not building the whole component means the need to understand well the partial solution).
- There is no formal method defined for testing and validation other than the general procedures from software life cycle.

## 7.3. Perspectives

At the beginning of working on this dissertation, this thesis assumed that because of the wonderful unifying properties of the MDA, the transition from object technology to modelling technology will bring huge conceptual simplification to the software engineering domain. Nevertheless, it surely did partially, but the road to model engineering takes time to fulfil all the promises. Therefore, I am fully aware that this is also particularly true for this thesis; although we have to admit that the today's software complexity and market pressures are bigger challenges to respond to, than object technology had to face before.

After first presentations of both, the approach and the prototype, [103] and [107], there were interesting insights and helpful feedbacks for improvement. This thesis already deals with some of them like: genericity integration (section 3.1 and 3.4.2), meta object protocol and component technology (section 3.2.1), a solution for domain driven development of applications (section 3.4.4), a methodology for describing models (section 3.4.5), software product lines support (chapter 4), migration of the prototype to standard tools (like Eclipse, see section 5.3), simple examples for the approach assessment (chapter 6).

This section presents the most important issues to address in the future and the order follows the same taxonomy as for contributions (see section 7.1).

### 7.3.1. SmartModels approach

Firstly, from the perspective of the SmartModels, it is important to continue the analysis of the meta-modelling approach itself in order to be more expressive and intuitive for designers and meta-programmers who are used with UML representations and their tool support ([89]). This approach is not sufficiently explicit or comprehensive to ensure consistent and interoperable implementations made by architects or programmers who use the meta-models.

Secondly, this research is mainly theoretical because the focus is on providing a way to model and use software product lines in a real model-oriented fashion, and not to implement industrial tools and support for architects. A new approach which tries from the very beginning to be industry-friendly will fall in a position to face many temptations to make compromises to favour the implementation instead of developing the semantics without this stress. Although I believe that this work is not far from answering this kind of challenges, more investigation is needed to offer a solution in this respect.

However, as an effect of the previous statement, SmartModels has to increase the semantics control over the behaviour of the modelled entities in the future applications. Even though I believe that this approach makes important steps in reducing the gap between the generic entities in the meta-level and their instances at the application level, the control upon the derived entities at execution time would be an interesting topic to study.

Then, as for any new approach, it is essential to keep on envisioning future trends on meta-modelling techniques for a new and better capture of the semantics. SmartModels, the approach developed by this thesis, does not advocate a pure top-down process based on models. It takes into account the practical methods and tools used by designers (especially the UML), but their concern is more bottom-up than handling generic concepts and writing abstract diagrams. Therefore, the current industry adopted methods have to be integrated in the approach as much as possible, but this dependency should be reduced to minimum when proposing a new one.

It is also important to provide more ways for semantics transformation of both models and applications when they evolve toward another model or application [105]. Through SmartModels' meta-level, the concepts allow classification and easy handling of model fragments which is useful is particular contexts. At this level, I believe would be interesting to envisage support for design patterns semantics-oriented and maybe support for recognition of pattern relevance.

### 7.3.2. SmartModels' support for SPL

The main application of SmartModels approach is to support SPL. I have decided to concentrate my efforts on applications which witness nowadays a rapid evolution and which require a high level of abstractization. In this way my interest is on the one hand in modelling them and on the other hand in their adaptation and reuse.

Firstly, I aim to further experiment the approach for the description of business models from various domains and their applications for building and customizing lines of products. In this way, I also hope to get more feedbacks from possible users in order to enlarge the applicability of SmartModels support for SPL and to ease the job of domain experts to describe a model ([42]).

Secondly, I believe that next important step is to provide a method to describe constraints on SmartModels entities based on OCL standard ([82] and [114]). I am thinking of a formal method which is intended to help developers to write invariants and pre/post conditions to SmartModels actions attached to concepts (see section 3.4.2 on concept semantics) using OCL standard and then generating code through mapping third party generators according to the targeted platform.

Thirdly, I am in search of new possible opportunities for product derivation (a possible trail may be using semantics redefinitions - see section 3.4.3). SmartModels approach is part of DDD trend on modelling which raises the level of abstraction beyond classical programming by offering the possibility to define domain specific concepts using visual modelling editors. Then the applications are written directly using these concepts. In this way, final products and their derivates can be generated automatically from these high-level specifications. If I succeed to enrich the methods used for product derivation (the SmartModels basic entities, OCL incorporation and the DSL for actions), I can diversify exponentially the number of products: the granularity of their modelling and the control over the code generators.

The last point, but not the least important, is the suggestion to do more investigation in order to advance the acceptance and viability of *model patterns*. We have all acknowledged the added value brought by design patterns in OOP and for now I make the assumption that I can extract from model patterns applied to the management of the meta-information of a specific domain's entities at least the same benefit.

### 7.3.3. SmartFactory prototype

SmartModels initiative has introduced a new approach for organizing the domain specific application development into different models so portability and reusability can be obtained through the independence both between semantics and data model and then the description of applications.

Therefore, the next paragraphs present challenges which I believe lie ahead SmartFactory. Although, the current state of the SmartFactory solves partially some of the paradigms, there are still important ideas to follow up.

Firstly, I will continue to seek to keep the prototype updated in order to support SmartModels approach improvements. Among them, I would mention as next important investments the previsioned OCL tool support and the DSL for customizing generic entities (see section 3.4.2) and for describing features provided by visitors when generating an application based on a model (see section 3.4.4).

Secondly, I endeavour to simplify the design of SmartFactory as a whole solution by adding on top a manager for a better automation between its modules: meta-data model editor, model transformer and code-generator (see section 5.4). The goal is to implement the possibility to generate code directly from the editor and let the editor to control also the code. Thus, the transformation phase should become invisible for the programmers, although I still consider to serialize the Eclipse compliant models for the sake of the openness of the experiments study.

Then, it is my commitment to improve the quality and the percentage of the source code automatically generated based on standard tools (like projects from Eclipse environment [35]). SmartFactory prototype aims to provide a framework where users can manipulate extensible models to ease the development of their applications integrating as much as possible generative programming capabilities.

# References

[1] Interim report: ANSI/X3/SPARC study group on Data Base Management Systems 75-02-08. *FDT - Bulletin of ACM SIGMOD*, 7(2):1–140, 1975.

[2] J.L. Lawall A.F. Le Meur and C. Consel. Specialization Scenarios : A Pragmatic Approach to Declaring Program Specialization. *Higher-Order and Symbolic Computation*, 17(1):47–92, 2004.

[3] M. Anastasopoulos and C. Gacek. Implementing Product Line Variabilities. Technical report, Report IESE Report No. 089.00/E, Version 1.0, IESE, November 2000.

[4] Ken Arnold, James Gosling, and David Holmes. The Java Programming Language, Third Edition. Addison-Wesley, 2000.

[5] Colin Atkinson, Joachim Bayer, Christian Bunse, Eric Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jorgen Wust, and Jorg Zettel: *Component Based Product Line Engineering with UML*. Addison-Wesley, 2002.

[6] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-Based Product Line Development: The KobrA Approach. In P. Donohoe, editor, *Proceedings of theFirstSoftware Product Line Conference*, pages 289–309, 2000.

[7] I. Attali, C. Courbis, P. Degenne, A. Fau, J. Fillon, D. Parigot, C. Pasquier, and C. Sacerdoti Coen. SmartTools: a Development Environment Generator based on XML Technologies. In *Proceedings of the Workshop on XML Technologies and Software Engineering at ICSE'01*, Toronto, Canada, May 2001.

[8] Felix Bachmann and Paul C. Clements. Variability in Software Product Lines. Technical report, CMU/SEI-2005-TR-012, 2005.

[9] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating Product-Lines Of Product-Families. *ase*, 00:81, 2002.

[10] Joachim Bayer. Introducing Separation of Concerns to Product Line Engineering. *GCSE'00 Young Researchers Workshop*, June 2000.

[11] Joachim Bayer. Towards Engineering Product Lines Using Concerns. *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, June 2000.

[12] B. W. Boehm. A Spiral Model of Software Development and Enhancement. 21:61–72, 1988.

[13] G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, 1991.

[14] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. The Object Technology Series. Addison-Wesley Publishing Co., October 1998.

[15] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In Frank van der Linden, editor, *PFE*, volume 2290 of *Lecture Notes in Computer Science*, pages 13–21. Springer, 2001.

[16] A. Capouillez, R. Chignoli, P. Crescenzo, and Ph. Lahire. Hyper-généricité pour les langages a objets : le modele OFL. In R. Godin and I. Borne, editors, *Actes de LMO'01, Conférence Nationale sur les Langages et Modeles a Objets. Publiés dans la revue L'objet : Logiciels, Bases de données, Réseaux*, volume 7(1-2/2001), pages 63–78, Le Croisic, France, janvier 2001. éditions Hermes Science Publications.

[17] A. Capouillez, P. Crescenzo, and Ph. Lahire. Le modele OFL au service du métaprogrammeur - Application a Java. In M. Dao and M. Huchard, editors, *Actes de LMO'2002, Conférence Nationale sur les Langages et Modeles a Objets. Publiés dans la revue L'objet : Logiciels, Bases de données, Réseaux*, volume 8(1-2/2002), pages 11–24, Montpellier, France, janvier 2002. Editions Hermes Science Publications. ISSN : 1262-1137 ; ISBN : 2-7462-0403-7.

[18] A. Capouillez, P. Crescenzo, and Ph. Lahire. OFL: Hyper-Genericity for Meta-Programming - An Application to Java. Research Report I3S/RR–2002-16–FR, I3S laboratory (UNSA/CNRS), Sophia-Antipolis, France, April 2002.

[19] C.-B Chirila, Ph. Lahire, D. Pescaru, and E. Tundrea. A Better Representation for Class Relationships in UML Using OFL Meta-information. In *AQTR 2004, International Conference on Automation, Quality & Testing, Robotics*, Cluj-Napoca, Romania, May 2004.

[20] Matthias Clauß. Generic Modeling Using UML Extensions For Variability. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, October 2001.

[21] Matthias Clauß. Modeling Variability with UML. GCSE 2001 Young Researchers Workshop Program, 2001.

[22] Crimson Consulting. *Total Costof Ownership Case Study: Large-Scale CRM System*. IBM 2001.

[23] P. Crescenzo. *OFL : un modele pour paramétrer la sémantique opérationnelle des langages a objets - Application aux relations inter-classes*. These de doctorat, Université de Nice-Sophia Antipolis, France, décembre 2001.

[24] P. Crescenzo and Ph. Lahire. Using both Specialisation and Generalisation in a Programming Language: Why and How? In J.-M. Bruel and Z. Bellahsene, editors, *Advances in Object-Oriented Information Systems : OOIS 2002 - Workshops proceedings*, LNCS(2426), pages 64–73, Montpellier, France, September 2002. Springer Verlag.

[25] Pierre Crescenzo and Philippe Lahire. Une approche pour améliorer la réutilisabilité des modeles métiers. In *Actes de la 2eme Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005)*, pages 51–73, Lille, septembre 2005.

[26] K. Czarnecki and W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, June 2000.

[27] Krzysztof Czarnecki and John Vlissides. *Domain Driven Development papers*. Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2003.

[28] P. Desfray. *Object Engineering, the Fourth Dimension*. Addison-Wesley Publishing Co., 1994.

[29] Deepak Dhungana. Integrated variability modeling of features and architecture in software product line engineering. *ase*, 0:327–330, 2006.

[30] W. El-Kaim. Managing variability in the LCAT SPLIT/Daisy model. In *Product Line Architecture Workshop SPLC1*, August 2000.

[31] W. El-Kaim. System Family Architecture Glossary. Technical report, ESAPS Project, 2001.

[32] D. Flanagan. *Java in a Nutshell: a Desktop Quick Reference*. O'Reilly, 3rd edition, December 1999.

[33] O. Flege. System Family Architecture Description Using the UML. Technical report, IESE-Report No. 092.00/E, IESE, December 2000.

[34] Oliver Flege, Joachim Bayer, and Cristina Gacek. Creating Product Line Architectures. In Frank van der Linden, editor, *IW-SAPF*, volume 1951 of *Lecture Notes in Computer Science*, pages 210–216. Springer, 2000.

[35] Eclipse Foundation. Eclipse Environment, 2004.

[36] Eclipse Foundation. Eclipse Modeling Framework Project (EMF), 2008.

[37] Steven D. Fraser, James Gosling, Anders Hejlsberg, Ole Lehrmann Madsen, Bertrand Meyer, and Guy L. Steele Jr. Celebrating 40 years of language evolution: Simula 67 to the present and beyond. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA Companion*, pages 1021–1023. ACM, 2007.

[38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[39] Michel Gauthier, editor. *Ada-Europe '93, 12th Ada-Europe International Conference, Ada Sans Frontieres, Paris, France, June 14-18, 1993, Proceedings*, volume 688 of *Lecture Notes in Computer Science*. Springer, 1993.

[40] A. Goldberg and K. Rubin. *Succeeding with Objects: Decision Frameworks for Project Management*. Addisson-Wesley, 1995.

[41] Hassan Gomaa. Object Oriented Analysis and Modeling for Families of Systems with UML. In William B. Frakes, editor, *ICSR*, volume 1844 of *Lecture Notes in Computer Science*, pages 89–99. Springer, 2000.

[42] Hassan Gomaa. Designing Software Product Lines with the Unified Modeling Language (UML). In Robert L. Nord, editor, *SPLC*, volume 3154 of *Lecture Notes in Computer Science*, page 317. Springer, 2004.

[43] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, 3 edition, June 2005.

[44] Martin L. Griss. Implementing product-line features with component reuse. In *International Conference on Software Reuse*, pages 137–152, 2000.

[45] Object Management Group. Model Driven Architectures homepage, 2003. http://www.omg.org/mda.

[46] W3C Working Group. *Web Services Architecture*. World Wide Web Consortium, February 2004.

[47] Günter Halmans and Klaus Pohl. Communicating the variability of a software-product family to customers. *Inform., Forsch. Entwickl.*, 18(3-4):113–131, 2004.

[48] W. Harrison and H. Ossher. Subject-Oriented Programming - A Critique of Pure Objects. In *proceedings of OOPSLA'93*, pages 411–428, Washington, D.C., United States, September 1993. ACM Press.

[49] George T. Heineman and William T. Councill. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[50] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[51] Robert Howard. The Eiffel Programming Language. *Dr. Dobb's J.*, 18(11):68-73, 1993.

[52] Apple Inc. Objective-C Programming Language. Technical report, 2008.

[53] M. A Jackson. *System development (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1983.

[54] I. Jacobson, M.L. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addisson-Wesley, 1997.

[55] I. Jacobson, M.Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addisson-Wesley, 1992.

[56] Michel Jaring and Jan Bosch. Variability dependencies in product family engineering. In Frank van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 81-97. Springer, 2003.

[57] Jean-Marc Jézéquel. Reifying variants in configuration management. *ACM Transaction on Software Engineering and Methodology*, 8(3):284-295, July 1999.

[58] Isabel John, Dirk Muthig, Peter Sody, and Enno Tolzmann. Efficient and systematic software evolution through domain analysis. *re*, 00:237, 2002.

[59] Jean-Marc Jézéquel. Model-Driven Engineering with Contracts, Patterns and Aspects. *Tutorial Program of AOSD 2003: 2nd International Conference on Aspect-Oriented Software Development*, March 2003.

[60] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[61] Alan C. Kay. The Early History of Smalltalk. In *HOPL Preprints*, pages 69-95, 1993.

[62] S. E. Keene. *A programmer's guide to object-oriented programming in Common LISP*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[63] Barry Keepence and Mike Mannion. Using patterns to model variability in product families. *IEEE Software*, 16(4):102-108, 1999.

[64] G. Kiczales, J. Des Rivieres, and D.G. Bobrow. *The Art of the MetaObject Protocol*. MIT-Press, 1991.

[65] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ.

[66] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP '97 Object-Oriented Programming 11th European*, volume 1241 of *Lecture Notes in Computer Science*, pages 220-242. Springer-Verlag, New York, NY, June 1997.

[67] P. Clements L. Bass and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[68] Philippe Lahire and Laurent Quintian. New perspective to improve reusability in object-oriented languages. *Journal Of Object Technology (JOT)*, 5(1):117-138, 2006.

[69] Roberto E. Lopez-Herrejon and Don Batory. Using AspectJ to implement product-lines: A case study. Department of Computer Sciences, The University of Texas, 2003.

[70] A. Maccari and A. Heie. Managing infinite variability. July 2003.

[71] A. Maccari and A-P. Tuvinen. System family architectures : Current challenges at Nokia. *F. van der Linden, IW-SAPF, Springer*, LNCS, 2000.

[72] Mike Mannion. Organizing for software product line engineering. *step*, 00:55, 2002.

[73] B. Meyer. *Object-Oriented Software Construction.* Professional Technical Reference. Prentice-Hall, 2nd edition, 1997.

[74] Georges Gardarin Mokrane Bouzeghoub and Patrick Valduriez. *Les Objets*. Eyrolles, 1997.

[75] Mark Nissen, Magdi Kamel, and Kishore Sengupta. Integrated analysis and design of knowledge systems and processes. *Inf. Resour. Manage. J.*, 13(1):24–42, 2000.

[76] L. Northrop and P. Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2002.

[77] Linda M. Northrop. A framework for software product line practice. In Ana M. D. Moreira and Serge Demeyer, editors, *ECOOP Workshops*, volume 1743 of *Lecture Notes in Computer Science*, pages 365–366. Springer, 1999.

[78] OMG. *UML Profile for Enterprise Distributed Object Computing Specification. Technical Report OMG Adopted Specification ptc/02-02-05*. Object Management Group (OMG), February 2002.

[79] OMG. *UML Profile for Schedulability, Performance, and Time Specification, Technical Report OMG Adopted Specification ptc/02-03-02*. Object Management Group (OMG), March 2002.

[80] OMG. *Unified Modeling Language Specification (UML) - Version 1.5.* Object Management Group (OMG), March 2003. Version 1.5.

[81] OMG. *Unified Modeling Language (UML) Infrastructure - Final Adopted Specifcation*. Object Management Group, September 2003. Version 2.0.

[82] OMG. *Unified Modeling Language (UML) OCL - Final Adopted Specification*. Object Management Group, October 2003. Version 2.0.

[83] OMG. *Unified Modeling Language (UML) Superstructure - Final Adopted Specification*. Object Management Group, August 2003. Version 2.0.

[84] OMG. *Meta-Object Facility (MOF) Core - Final Adopted Specification*. Object Management Group, March 2004. Version 2.0.

[85] OMG. *XML MetaData Interchange (XMI) - XML Schema*. Object Management Group, May 2004. Version 2.0.

[86] OMG. *Model-Driven Architecture - (MDA)*. Object Management Group (OMG), July 2006.

[87] J. Palsberg and C.B. Jay. The Essence of the Visitor Pattern. In *COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998. IEEE Computer Society.

[88] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[89] D. Pescaru. *Bridging the Gap between Object Oriented Modeling and Implementation Languages Using a Meta-Language Approach*. Ph.D Thesis, "Politehnica" University of Timisoara, Timisoara, Roumanie, December 2003.

[90] CAFE project. Technical report, http://www.esi.es/en/projects/cafe/cafe.html, 2001.

[91] ESAPS project. Technical report, http://www.esi.es/esaps/, 1999.

[92] FAMILIES project. Technical report, http://www.esi.es/en/projects/families/, 2003.

[93] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[94] C. Simonyi. The Death of Programming Languages, the Birth of Intentional Programming. Technical report, Microsoft, Inc., September 1995.

[95] Objecteering Software. http://www.objecteering.com/.

[96] R. Soley and OMG. *Model driven architecture (2000)*. OMG Group, 2000.

[97] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, 8th edition, 2007.

[98] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., 3rd edition, 1997.

[99] Mikael Svahnberg and Jan Bosch. Issues concerning variability in software product lines. In Frank van der Linden, editor, *IW-SAPF*, volume 1951 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 2000.

[100] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 2002.

[101] H. Tardieu, A. Rochfeld, and R. Coletti. *La Methode Merise*. Les Editions d'Organisation, Paris, 1983.

**[102] E. Tundrea**, Ph. Lahire, D. Parigot, C. Courbis, , and P. Crescenzo. An Attempt to Set the Framework of Model-Oriented Programming. In *CONTI 2004, 6th International Conference on Technical Informatics*, volume 3 of *Transaction on Automatic Control and Computer Science*, pages 71–76, Timisoara, Romania, May 2004. Scientific Bulletin of "Politehnica" University of Timisoara.

[103] Emanuel Tundrea. Smartmodels - an Approach for Developing Software Based on Models: Rules, Prototypes and Examples. Technical report, Politehnica University of Timisoara, February 2006.

**[104] Emanuel Tundrea**, SmartModels – An MDE Platform for the Management of Software Product Lines. In *Proceedings of 2008 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR 2008)*, Volume 3, 22-25 May 2008 Page(s):193 - 199, Cluj-Napoca, Romania.

**[105] Emanuel Tundrea**, Pierre Crescenzo, and Philippe Lahire. La généricité paramétrée au service des modeles métiers. In R. Rousseau and C. Urtado, editors, *Actes de LMO'2006, conférence nationale sur les Langages et Modeles a Objets.*, pages 151–166, Nîmes, France, mars 2006. Editions Hermes Lavoisier.

**[106] Emanuel Tundrea**, Philippe Lahire, Dan Pescaru, and Ciprian-Bogdan Chirila. Smartmodels – an Attempt to Set the Framework of Model-Oriented Programming. In *Conti 2004 - The 6th International Conference on Technical Informatics, Scientific Bulletin of "Politehnica" University of Timisoara, Romania, Transactions on Automatic Control and Computer Science*, Timisoara, Romania, May 2004. Editura Politehnica, Vol. 49(63) 2004 No. 1,2,3,4 / ISSN 1224-600X.

**[107] Emanuel Tundrea**, Philippe Lahire, Dan Pescaru, and Ciprian-Bogdan Chirila. Smartfactory – a Prototype for Model Oriented Software Engineering Based on Eclipse Platform. In *Proceedings of International Conference on Technical Informatics - CONTI'2006*, vol. 2, pp. 71-76, ISBN (10): 973-625-319-8 ISBN (13): 978-973-625-319-5, Timisoara, Romania, June 8-9, 2006.

**[108] Emanuel Tundrea**, Philippe Lahire, Dan Pescaru, and Ciprian-Bogdan Chirila. Smartmodels - a Framework for Generating On-Line Learning Software Solutions. In *Proceedings of the 12th International Conference Netties 2006*, pp. 69-75, ISBN (10): 973-638-262-1 ISBN (13): 978-973-638-262-8, Orizonturi Universitare Publishing House, Timisoara, Romania, September 6-9, 2006.

[109] Sebastian Tyrrell. The many dimensions of the software process. *Crossroads - The ACM Student Magazine*, Windows Programming Summer 2000 - 6.4, 2000.

[110] Frank van der Linden. Software Product Families in Europe: The ESAPS & CAFÉ projects. *IEEE Software*, 19(4):41-49, 2002.

[111] Frank van der Linden, editor. *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, volume 3014 of *Lecture Notes in Computer Science*. Springer, 2004.

[112] T. van der Maßen and H. Lichter. Modeling Variability by UML Use Case Diagrams. *International Workshop on Requirement Engineering for Product Line (REPL02)*, September 2002.

[113] Rob C. van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78-85, 2000.

[114] Gilles Vanwormhoudt. Précision et validation de métamodeles avec EMF et OCL. In Ph. Collet and Ph. Lahire, editors, *Actes des Journée Objects Composants et Modeles (OCM) du GDR ALP*, pages 19-28, Berne, Mars 2005.

[115] Angi Voß. C++, Common LISP/CLOS, Eiffel oder Smalltalk? *KI*, 9(2):35-44, 1995.

[116] Diana L. Webber and Hassan Gomaa. Modeling Variability in Software Product Lines with the Variation Point Model. *Computer Science Program*, 53(3):305-331, 2004.

[117] M.D. Weiss and C.T. Robert Lai. *Software Product-Line Engineering : A Family-Based Software Development Process*. Addison-Wisley, 1999.

[118] www.metamodel.com. Meta-Model, 2008.

[119] Tewfik Ziadi, Jean-Marc Jézéquel, and Frédéric Fondement. Product Line Derivation with UML. In *Proceedings Software Variability Management Workshop, Univ. of Groningen Departement of Mathematics and Computing Science*, February 2003.