Conceiving and Implementing a language-oriented approach for the design of automated learning scenarios

César Moura

Université des sciences et technologies de Lille

École Doctorale des Sciences Pour l'Ingénieur

Thèse

pour obtenir le grade de

Docteur de l'Université des sciences et technologies de Lille

Discipline: Informatique

Présentée et soutenue publiquement par

DE MOURA FILHO César Olavo

le 20 juin 2007

MDEduc: conceiving and implementing a language-oriented approach for the design of automated learning scenarios

Directeur de la thèse :

Pr. Alain DERYCKE (USTL)

Membres du jury :

Président : Pr. Jean-Luc DEKEYSER (USTL)

Rapporteur : Pr. Eric BRUILLARD (ENS-Cachan)

Rapporteur: MCF Thierry NODENOT (Univ. de Pau et des Pays de l'Adour)

Examinateur: Pr. Mauro OLIVEIRA (CEFET-CE - Brésil)

Co-encadrant: MCF Yvan PETER (USTL)

This is a French thesis. Even if it has been written in English and by a Brazilian, it remains a French thesis, for it is impregnated with the French academic tradition and - why not? – intuition, without which it would be something else. Thus, I dedicate this work to the country who has welcomed me along with my family during the last four years (even if it has the bad taste to often beat Brazil in World Cups…)

Acknowledgements

I would like to thank all the people who somehow helped me along the winding and strenuous road that led to this manuscript. First and foremost, I'd like to say a big thank to my family (Matias, Pedro and Joélia) and especially to Bianca, who had to put up with the most unavailable, crusty, irascible, cantankerous and stubborn César ever, particularly in the last year of this thesis.

It has been a great pleasure to work with my advisor, Professor Alain Derycke, who has insisted on instilling in me the rigors of the scientific research (though I'm not sure I managed to "get the whole picture" yet). I personally don't think there are many people out there that could guide me with so much competence through all the areas I had to dig into along this thesis.

I'd also want to thank the reviewers, Professor Eric Bruillard and Professor Thierry Nodenot, who so kindly and promptly accepted to read and to comment on this manuscript, providing many insightful remarks that have already been incorporated to this work and that will surely be considered in my future research; to Professor Jean-Luc Dekeyser, for kindly accepting to chair the *viva voce* panel and to Professor Mauro Oliveira, an old time friend who motivated me to pursue this PhD (I still don't know if I'm going to forgive him one day for this…) and for examining this *mémoire*.

I would like to thank Yvan Peter, the co-advisor of this thesis, always willing to help me whenever I stumbled upon the difficulties of the computer domain. I'm also grateful to the members of the NOCE Team and, in particular, to Xavier Le Pallec and Pierre-André Caron, with whom my relations crossed the limits of the Trigone Laboratory, making the stay in France still more pleasant for my family and me.

I Thank John Hicks for the good moments spent together, whether discussing computer related themes or playing the guitar. I hope we will be able to produce other musical and scientific pieces together in the future. I'm also profoundly indebted to Joël Cheuoua, one of the rare examples of people ready to go a long way to help a strange - and asking nothing in return. Thank him for seeing me through some of the most advanced - and highly undocumented - features of Eclipse, EMF, JET, etc.

Last, but not least, I give thanks to the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), for providing the necessary funding for this work and to the Centro Federal de Educação Tecnológica do Ceará (CEFET-CE), in particular, to the fellows from the Department of Telematics, for assuming my duties while I was absent; a special thank to Mariângela (yes, in Brazil last names are almost useless…), my first time rescuer, whenever Brazilian bureaucracy threatened to swallow me up.

Abstract

Aiming at developing learning applications, the e-learning communities have recently made a profound switch in their modus operandi. Instead of following prescriptive processes leading to turn-key applications reflecting an envisaged learning scenario, as in traditional ISD processes, learning designers now make use of Educational Modeling Languages (EML), provided by standard setting bodies along with the required tooling, so they can build their own applications. Being at close range to practitioners, learning designers will be able to build tailored applications in a fraction of the time required by traditional ISD and in a much more flexible approach - since no programming is needed.

However this flexibility is limited. EMLs present a fixed vocabulary, permitting its elements to be arranged to create an instance of a scenario, but disallowing the addition of new concepts - in case existing ones do not satisfy. Since EMLs, as epitomized by the IMS-LD specification, intend to capture all aspects of educational phenomena, they tend to become either too complex or too inexpressive - in the first case becoming unwieldy for tackling any particular scenario and in the second, failing to conveniently express any reasonably rich scenario. Furthermore, it is claimed on this thesis that any particular EML cannot represent but a single aspect of learning scenarios, which in real life are multi-dimensional and always open to diverse interpretations by different people.

So, this thesis sustains that the single EML approach is fundamentally flawed, and proposes an alternative one, called multi-EML, which empowers learning designers with the possibility of formulating the very conceptualization of their problems, by allowing them to create their own EML reflecting their specific requirements - as opposed to just rearrange the concepts of a fixed language. In sum, the Multi-EML approach enables:

1. capturing the knowledge of domain experts in informal notations (particularly using design patterns for education, or pedagogical patterns);
2. starting from these original descriptions, to build formal models (or EMLS) representing their specific requirements;
3. generating full-fledged applications driven by the formal models.

A prototype has been developed that implements these ideas. To permit automated scenarios to be as expressive as possible of real life learning experiments, the prototype enables those who best know their specific fields -i.e. the domain experts - to take part in the development efforts, by participating in any of the three steps above.

Particularly for the third step - which corresponds to developing software applications -, this can only be possible if we succeed in abstracting out all computer related idioms and in offering practitioners a programming environment that requires only the knowledge of their own domains, an idea that Martin Fowler calls "lay programming".

Taking all three steps, designers will be allowed to go from an informal description to an executable application, which will be automatically generated. This is only feasible thanks to recent developments in software engineering, and particularly to the Language Oriented Programming (or LOP), a new programming paradigm that counts technologies like OMG's Model Driven Approach, the Eclipse Modeling Framework and Microsoft's Software Factories amongst its most famous implementations.

As the result of bringing the power of LOP into the world of learning application design, we expect a significant increase in the expressiveness of the scenarios and a reduction in the development life-cycle of learning applications. It is envisaged that the multi-EML approach will afford learning applications to be created as a direct extension of the user experience.

Résumé

Cette thèse a pour sujet la conception de scénarios pédagogiques destinés à l'e-formation. Afin de faciliter les échanges de matériaux décrivant des stratégies pédagogiques, la communauté s'est récemment mobilisée pour proposer un langage standard suffisamment générique pour permettre la représentation de n'importe quel scénario, indépendant même du paradigme éducationnel sous-jacent. Appelé génériquement Educational Modeling Language (EML), ce type de langage engendre une nouvelle façon de concevoir des EIAH, en s'éloignant du traditionnel Instructional System Design, une fois que, au lieu de proposer une application finie, les EML proposent un modèle conceptuel standard, une notation pour l'exprimer et des éditeurs et *frameworks*, laissant aux concepteurs finaux la tâche de créer leurs propres « applications ». Les EMLs permettent alors la création et exécution d'instances de scénarios, dans une approche plus ouverte et flexible, augmentant, ainsi, les possibilités d'adaptation des applications résultantes aux besoins des usagers.

Cette flexibilité reste pourtant limitée et, après quelques années de recherche, les EMLs commencent à montrer ses faiblesses. En fait, le langage choisi pour devenir le standard du domaine, le IMS-LD, s'est montré générique, certes, mais peu expressive, ne permettant pas une représentation fidèle des divers scénarios existants. C'est à dire, c'est aux usagers de s'adapter à la syntaxe et sémantique de cet standard.

Cette thèse part d'un constat quant aux difficultés du processus de conception lui-même, et aux risques de coupure qu'il peut y avoir entre pédagogues et développeurs de logiciels. Pour améliorer la capacité des équipes pédagogiques à pouvoir spécifier, et même implémenter, des scénarios pédagogiques, nous proposons une approche où c'est l'EML qui doit s'adapter aux besoins de l'usager. L'usager a la possibilité de créer son propre langage (ou ses propres langages), s'il en a besoin. En plus, un même scénario peut être décrit en même temps par des différents EMLs (ou modèles) respectant des différents perspectives - et même paradigmes - de chaque *stake holder*.

Cette approche, appelée multi-EML, est possible grâce aux avancées récentes du génie logiciel, telle l'Architecture Dirigée par les Modèles – l'implémentation la plus connue d'un nouvel paradigme de programmation surnommé Languages Oriented Programming (LOP), qui inclut encore d'autres implémentations.

Notre proposition réside dans la conception d'un environnement informatique « auteur », qui repose sur les principes des Languages Oriented Programming, en utilisant la plateforme ouverte ECLIPSE et, plus particulièrement son implémentation du LOP, l'Eclipse Modeling Framework (EMF). Ainsi, les concepteurs auront un outil qui leur permettra de créer des spécifications formelles décrivant les scénarios envisagés et d'en générer automatiquement des applications correspondantes, dans un processus qui démarre avec les descriptions informelles des experts du domaine.

Reconnaissant que les experts d'éducation - ceux qui mieux comprennent le domaine - ne sont pas nécessairement des informaticiens, l'environnement proposé, appelé MDEduc, fournit aussi un éditeur permettant de décrire un scénario dans une notation informelle, à savoir le pattern pédagogique, à partir de laquelle les modèles formels peuvent être dérivés. En plus, nous proposons de garder côte à côte et en coïncidence ces descriptions en langage informelles, et les descriptions plus formelles et normatives et d'offrir la possibilité d'effectuer des allers-retours à toutes les phases du cycle de vie du dispositif pédagogique.

MOTS-CLÉS : informatique, pédagogie, e-formation , conception des EIAH, modélisation, scénario pédagogique, démarche de conception par patrons, patrons pédagogiques, ingénierie dirigée par les modèles, normes et standards, langages de descriptions, environnement de conception logiciel, génie logiciel, plate-forme ECLIPSE et EMF.

“Bota força nessa coisa
Que se a coisa pára
A gente fica cara a cara
Com o que não quer ver”

Chico Buarque

Contents

List of figures

Chapter One: Introduction

1.1 Introduction

Technology plays an increasing role in education. Developments in Internet and multimedia technologies have multiplied the possibilities of the use of computers in classroom-based and most notably distant learning courses, themselves becoming more and more conflated with e-learning. In fact, e-learning is nowadays becoming a prosaic solution, especially in niches surrounding graduate certificates, people with reduced mobility, students from secluded locations and other specific cases that take advantage of the asynchronous pace of e-learning courses.

However, this rapid success comes at a price. Critics attest that instructional technology did not live up to its most grandiose promises (Clark, 1994; Cuban, 2001; Dillon & Gabbard, 1998; Fabos & Young, 1999), giving the impression that it "took off before people really knew how to use it" (Zemsky & Massy, 2003). Of three (unaccomplished) e-learning pledges declared by (Zemsky & Massy, 2003), one concerns directly this work: that the marriage of new electronic technologies and newly accepted theories of learning *will* yield a revolution in pedagogy itself.

Acknowledging this fiasco, researchers in the domain of instructional technology have been devoting considerable time and effort towards improving the quality of computer programs used in education and, as a consequence, new approaches have been devised to cope with the design and implementation of learning applications - as will be shown throughout this thesis. However, even if some progress has been made, it is necessary to recognize that the promise that computers and e-learning will change the way we teach has still not been met – "and by a long shot".

This dissertation represents a new endeavor in this direction. If successful, it will be an alternative to the existing approaches for designing and implementing learning applications.

1.2 Problem context

As stated in the previous section, researchers are coming up with new ways of designing and developing learning applications. After many years using Instructional System Design (ISD) techniques, developers recently changed the approach radically. Instead of delivering full fledged applications to end users, as per ISD precepts, they now offer metamodels to domain experts - accompanied by a framework composed of an Application Programming Interface, editors, runtime engine, etc. – with which they can assemble their own learning applications. These metamodels are called Educational Modeling Languages (EML, 2000) and are used to formally represent the usual elements we find in educational events. More details of these different techniques will be given in Chapter Two.

In fact, this is possibly the last time I will use the expression "design learning applications" in this thesis, because I consider that what has to be designed and implemented is a *learning scenario*, of which the learning application is just one element. In that case, EMLs are formal languages used for designing learning scenarios.

The usual connotation of *learning scenario*, as found in the literature, is, roughly, the *specification of a workflow of learning activities*. I ignore the contingencies of this metonymical reduction being applied to this important concept and why it is widely accepted. In fact, even though the current work has begun in accordance with the mainstream signification, the usual vicissitudes of exploratory research have meant it soon became unfit for the needs of this work.

For the purpose of this thesis, then - though this concept will be more precisely discussed in the ensuing chapters - a learning scenario is a description of a learning facilitating experiment, which specifies how actors, applications, learning concepts, operational strategies and many other elements configuring the context (like motivations, sets of values and beliefs, goals, geographic and social factors, etc.) come into play during the experiment. Thus, if we consider this definition of learning scenario, we soon apprehend that *workflows of activities* represent only one perspective of a much more complex whole. Moreover, because these descriptions are formal specifications that will drive applications, I refer the expression *automated learning scenarios*.

The research into languages used for describing learning scenarios, or EMLs, has been centered on the concept of one standard language, powerful enough to represent every possible learning scenario. A first serious move in this direction was the adoption of the Educational Modeling Language - originally developed at the Open University of the Netherlands – in 2003 by the IMS Global Learning Consortium, which gave rise to the Learning Design specification (IMS-LD, 2003).

However, such a unary approach, proposing a single EML to represent all learning scenarios, seems fundamentally flawed. This is due firstly to the fact that education as a whole is too complex a domain to fit in a single hierarchical model; secondly, real teaching/learning experiments are multi-dimensional phenomena, admitting as many interpretations as there are observers, each one possibly drawing on a different paradigm. Thus, even if there were compelling arguments for the superiority of any single representation – an assumption behind the single EML approach -, it would certainly not be better than having multiple and complimentary models eliciting the different perspectives present in real situations.

Thus, this thesis proposes an approach for tackling the question of the design of automated learning scenarios that takes account of this multi-dimensional aspect of real teaching/learning situations. It does so by enabling learning designers and instructors alike to represent their knowledge in formal descriptions and subsequently to generate software programs conforming to these descriptions.

1.3 Thesis Context

Addressing this proposition required the concurrence of different bodies of knowledge. One area that provided important insights for this thesis was the *Theory of Design*. Since we are talking about the creation of scenarios, we have to understand what it means to create things – and the Theory of Design affords an interesting conceptual framework within which such an understanding can be obtained.

Thus, this work positions itself in the blurred line that divides design and science. That is, it is a thesis in Computer Science – hence guided by assertive rules - but one that draws inspiration from the Theory of Design, a locus governed by modal logic and in which some

scientific tenets, like universality and reproducibility must yield slightly, making room for other goal-based principles.

Thus, while using Computer Science and Theory of Design as tools, this thesis has its "business domain" in Education. In fact, education itself belongs in the non-assertive world of design. Being an archetypal creation of the human intellect, education poses questions that cannot be solved in the light of science:

> The process of education is not a natural phenomenon of the kind that has sometimes rewarded scientific investigation. It is not one of the givens in our universe. It is man-made, designed to serve our needs. It is not governed by any natural laws. It is not in need of research to find out how it works. It is in need of creative invention to make it work better (Robert Ebel *apud* (Farley, 1982)).

In trying to cope with design based principles, this thesis deals with what (J. Coplien, 2000) dubs "the most difficult issue in contemporary Computer Science," when you have to leave formal, deterministic mathematical and analytical essence of programming and search beyond it, for the principles guiding human creation, an arena where mathematical principles do not afford conclusive answers.

For a multidisciplinary thesis, I would certainly need to work with a multidisciplinary team, and my research took place in the NOCE Team (Nouvelles Organisations pour la Coopération et l'Éducation) of the Trigone Laboratory, which investigates how computers can more efficiently mediate between *cooperation* and *education*. In particular, this work puts more emphasis on the educational component. Among the specific technological subjects addressed at in NOCE's projects, we find themes as different as Human-Computer Interaction, Workflow Systems, Middleware, Software Engineering (and more notably Model Driven Engineering, as in the case of this thesis), etc., but all of them at the service of the two eminently social aspects mentioned above.

Other works exist in the NOCE team that have similar interests. For example, (Caron, 2007) proposes to capture the metamodels underlying some of the well known Learning Management Systems (LMS) and to expose them to learning designers, so they can conceive learning scenarios, with the help of a graphical editor. Subsequently, the designed scenarios are exported, through web services, to their originating LMSs, causing these platforms to be populated with courses, classes, etc. without having to enter data manually. Also, the

mentioned graphical scenario editor, called ModX (LePallec, 2002), was itself another contribution by the NOCE team to the learning design community, and has been regularly used by several research teams from different universities in France.

1.4 Outline of this Thesis

The remainder of this dissertation is outlined as follows:

Chapter Two scrutinizes the question of design. Its main objective is to provide an answer to the question: "what does it mean, *to design*?" Gradually, it applies the answer to *learning scenarios*. Thus, a *learning scenario* is the unit of design to be considered in this thesis. The chapter then goes on to describe different manners of designing these learning scenarios, in both formal and informal representations, providing examples.

Chapter Three suggests the criteria by which one might measure the quality of formal representations, explaining along the way why single hierarchical representations will inescapably fall short of conveniently addressing complex domains, and in particular, the learning design domain; this chapter makes the case for using multiple educational modeling languages to represent learning scenarios, advancing the approach that will address the shortcomings of using single models.

Chapter Four discusses the technology that will allow us to realize the approach proposed in Chapter Three. It presents the Language Oriented Programming (LOP), a novel programming paradigm that enables programmers to conceive software applications while taking into consideration only sets of high-level concepts used by the experts from the domain in question. Nominally, I am interested in this thesis for concepts in the domain of education. Thus, by factoring out computer related idioms, LOP implementations bring the programming task within the reach of domain experts, enabling so-called *lay programming*. After presenting some LOP implementations, a choice is made for the Eclipse/EMF solution and a justification is laid out. Technical details of this particular choice are provided in Appendices A and B. Other LOP implementations not covered in this chapter are shown in Appendix C.

Chapter Five describes a prototype application that implements the approach advanced in Chapter Three, illustrating its use with a concrete case. Chapter Six wraps up the thesis, tying together the claims posed along the work and proposing further trails to be investigated in future works.

Chapter Two: The Design of Learning Scenarios

This thesis proposes a new approach for the **Design of Learning Scenarios**. *Design* is a word that has long entered the popular consciousness and, as a consequence, now lends itself to as many interpretations as there are people uttering it – and as many more as listeners. *Learning scenario*, on the other hand, is a much more recent and use restricted expression. However, if there are less possible interpretations for it, this is because there has been the convergence toward a mainstream definition, as pointed out in the introduction. So, first of all, it is important to clearly contextualize, for the scope of this thesis, the terms "design" and "learning scenario" and to precisely specify our understanding - from which our contribution will derive - of these two far reaching terms. This is the main objective of this chapter.

2.1 Introduction

Communities in their everyday practice specialize terms that have very precise and specific semantics, adapted to their specific needs. Even common terms with time will acquire specialized meanings in these communities of practice that depart significantly from their everyday usage. The collection of such terms and semantics they bear form the community's language. It is this language that will mold what can be represented, what can be designed, what can be built, and ultimately, what can and what cannot be said.

Similarly, in addition to languages, communities also share practices. That is, once a vocabulary is agreed upon, experts develop efficient and sometimes proven ways of doing their activities. These may range from rote procedures – rather inflexible sequences of action specified in advance or in the abstract, usually serving as a way into skills for novices (Robinson, 1997) –, to more elaborate and flexible formalisms, like scenarios, design patterns, etc.

What these two approaches, i.e. language generation and practice build-up, have in common is that both can be viewed as distinct but related forms of design. This duality is not exclusive from the design theory, though, and reflects two broader worldviews, which I will call *entity-centric* and *process-centric*. However, before I explore this subject any further, I provide, as a

primer, a concise description of this duality from a philosophical viewpoint, extracted from the Wikipedia's article on *Process Theory* and reproduced in the footnote[1]. While it is never good practice to quote large portions of text, in this case I consider that it will spare the reader from digressing into an external source for a second view on an overarching subject that is central to this thesis but of which I treat only a particular case.

Thus, applying these two worldviews to design means, when looking from an *entity-centric* perspective, that the designer's task is to identify, define, realize and verify all the required constituent parts of the designed artifact, aiming at some desired overall characteristics (concerning functionality, usability, reliability, customizability, etc.). It is an analytical approach whose main objective is to define *what* will be designed, giving a response in structural terms – that is why we could also call it *structure-centric*.

The *process-centric* perspective, on the other hand, will define *how* we are going to design that same artifact. It is an action that occurs usually later in time, after the entity-centric design has been achieved (even if provisionally). That's why the designer, at this stage, can envisage some more practical requirements, concerning, for example, the process of development, such as cost, duration of development, risks in development, aesthetic, etc.

---

[1] "Western science emerged out of philosophy. It is notable that ancient and Enlightenment era Western philosophy completely overlooked the power of process in producing effects. For instance, Plato imagined "forms" and the atomists imagined "atoms" (in their original Greek sense) as fully explaining reality in its "current state." The problem with such accounts of "current state" reality is that we are left with our theoretical entities to account for."

"In the 19th century, science began to part with this old "entity-centric" view in favor of processes. One of the earliest is Charles Darwin's theory of evolution. It was followed by the Big Bang theory and plate tectonics. In these theories, complex "current states" can be explained in terms of processes that occurred over time — generally evolving from simpler more primordial states. One of the advantages of process theories is that they avoid endless regress of explanations, as complex states arise from simpler states."

"Only very recently has this thinking begun to enter philosophy. Rather than accounting for experience through hypothetical entities and forces (such as matter and energy) philosophers are beginning to postulate an evolution of experience itself — resulting in fewer "working parts" and surprising degrees of explanatory power. What was taken to be the result of matter and energy (the effect as presented in human experience) is then simply reassigned as a piece in a perceptual process. This is still a new and radical view and is not as of yet the general consensus, but it does appear to be persisting as an alternative worldview."

However, it is important to note, while process-centric approaches fulfill an important practical role, they “do not fill the irreplaceable theoretical void”.

To summarize, while an entity-centric design defines what fundamental pieces can be considered in the activity of design, process-centric design describes practical ways of how these pieces can be strung together toward a *goal*. Furthermore, if the former is analytical, the latter is descriptive.

This two-sided approach, as defined here, is not a premiere in design-related domains. (Floyd, 1987), for example, has already reported it before for the discipline of Software Engineering, even if with different terminologies. To her, "the product-oriented perspective regards software as a product standing on its own, consisting of a set of programs and related defining texts" while the process-oriented perspective views software "in connection with human learning, work and communication, taking place in an evolving world with changing needs [and] emerging from the totality of interleaved processes of analysis, design, implementation, evaluation and feedback, carried out by different groups of people involved in system development in various roles."

Even if I insist on the goal-oriented nature of the process-centric design, it is also important to note that the entity-centric design, as any design effort, is also clearly goal-based. When communities conceive their languages, they are in fact working ways to solve their problems and such an activity presents an outstandingly teleological nature. Notwithstanding, it is more directed at a final cause, whereas process-centric design presents more immediate goals.

As mentioned before, these are distinct, but related forms of design. In fact there is some kind of interplay between both, such that changes operated on one will more often than not impinge on the other as well. A typical example is a language -- the archetypal representative of the entity-centric design -- being used to formally represent knowledge, which clearly influences how practices -- hence, process-centric design -- will be encoded with it. Certainly, a new concept added to a language will afford new ways of arranging its terms, allowing new practices to be formulated.

The influence of processes into the structures is also remarkable, though less direct. All the same, insights brought about during design *processes* may take designers to reconsider their languages. The difficulty or impossibility of arranging their terms toward a specific goal may

suggest changes to their languages – e.g., additions, modifications, exclusions – so as to adapt it to designers' needs and wants.

Usually, it is in the initial entity-based design that the overall possibilities of the product are to be defined. For this to happen, designers, limited by their ability to anticipate the future, must look well in advance -- for example, by conceiving ad-hoc process-based designs for a few notable situations -- in an effort to delimit the scope of their language. Conversely, process-designers do not necessarily use all pieces in a given design, since they focus on particular, fine-tuned solutions for specific situations, which may sometimes require only a fraction of existing parts.

However, even though in an ideal case entity-class designs are supposed to predict all possible consequences from the beginning, the most typical scenario is where they evolve gradually and consciously, as designers gain experience on the problem:

> Complex systems can evolve without a coherent master design—for example, cities and the Internet—but even in these cases, conscious design is at work in creating the individual pieces and relationships that make up the whole. (Winograd, 1996)

Though it is not uncommon, especially among beginners, to mix entity-centric and process-centric design approaches, there seems to be a clear reward in tackling them separately. Software designers, for example, learn soon the advantages of applying the Model-View-Controller[2] architectural pattern, which enforces the separation of the *model* – entity-centric portion of a program -, from the *controller* – the process-centric part, where the business logic is described.

For the remainder of this chapter I will discuss the question of design ("what does it mean to design, after all?") and illustrate this discussion with some exemplar approaches for design at large, and to the design in the domain of education

---

[2] The MVC pattern separates applications in three layers: the **Model** is the **domain**-specific representation of the information on which the application operates; the View renders the model into a form suitable for interaction, typically a UI element and the Controller processes and responds to events, typically user actions, and may invoke changes on the model (adapted from the Wikipedia article on MVC).

2.2 About Design

In his seminal book, *The Sciences of the Artificial,* (Simon, 1996) argues for the need of systematization of a body of knowledge whose main interest is not to explain natural phenomena as they *are* – something *natural sciences* already do well – but, instead, to account for the *design* of phenomena as something we think they *ought* to be, as a result of human volition and having some goal in mind:

> The engineer, and more generally the designer, is concerned with how things *ought* to be – how they ought to be in order to *attain goals,* and to *function.*

This is a breakthrough approach that Simon carves out as a new body of knowledge, which he calls the *sciences of the artificial*, weaning it off the natural sciences. That is, through it, a parallel world suddenly appears that has "its own logic, which is related to, but is quite distinct from, the logic of the natural world" (Dasgupta, 1992).

Simon's science of the artificial is the baseline I will use in this section to discuss *design*. Given the difficulty to add any original contribution to a subject so cogently constructed, this section will be more a compendium of ideas about design – or, most frequently, interpretations thereof - introduced by Simon and by some authors that adventured into expanding his concepts. That is why I am pleading guilty in advance to quoting him more often than I probably should.

One of the first ideas that stands out of Simon's work is that, while being used in different bodies of knowledge, design presents some underlying principles, independent of any specific domain, which calls for a subject of inquiry in itself, i.e. a kind of unified theory of design:

> While there are many distinct sciences of the artificial - civil, mechanical and electrical engineering, the chemical, metallurgical and process technologies, agriculture, computer science, organization theory, economic and social planning, architecture, etc. - there is one kind of intellectually nontrivial activity that is common to all and at the heart of each, viz., the activity of **design**. (Dasgupta, 1992)

The theory of design, so construed, is not necessarily congruent with all principles from natural sciences, i.e., it may not always satisfy all the scientific canons – e.g., of reproducibility, of generality, of testability, etc. When imbued with the design mindset, designers are willing to renounce, if needed, "analytic, formalizable" explanations in favor of

“partly formalizable, partly empirical” processes - even because scientific rigor will not provide all answers required by the design activity.

Yet it does not mean it is irrational either. When I say that a theory of design does not abide by the (natural) scientific rules, I do not mean that these are deliberately abandoned altogether or that artifacts have a “dispensation to ignore or violate natural law”, but that, instead, in a design, natural laws must now share the center stage with other purposeful, goal-based principles. In other words, with design, it may be pointless to meticulously aim at precision rigor or reproducibility if the goal is not ultimately reached. For example:

> Even if you can prove that a program satisfies a mathematical specification, there is no way to prove that the mathematical specification meets the real requirements of the system (Fowler, 2004).

And Fowler goes on to justify the lack of precision - being traded for usefulness – of the semantically loose graphic model of the Unified Modeling Language (UML):

> These [graphical modeling languages] may be informal, but many people still find them useful—and it is usefulness that counts (Fowler, ibidem).

That is, design is also about utility. Furthermore, design involves multiple choices between distinct satisfactory alternatives to attain a goal, what requires a kind of reasoning to which standard systems of logic (e.g. propositional and predicate calculi) are not particularly adapted, as they have been conceived for the assertive world typical of natural sciences. Simon provides some examples of “paradoxes” we may incur when attempting to apply imperative logic to modal-type situations, with their “shoulds” and “ougths”, common in the process of design.

According to (Gibbons & Rogers, 2007), Simon “portrays the controlling logic of design as the formation and exploration of a set of alternative solutions that satisfy a set of constraints and criteria and then selection of an alternative on the basis of a prioritizing rule”. Actually, it is exactly the problem of the search for the satisfactory alternatives that is, according to Simon, particularly vexing to assertive logic (Simon, ibidem, pp. 121-122).

In fact, Simon did make a great effort to couch his theory of design in scientific foundations, without which it would be reduced to a “cook-book” approach. In addressing the problem he pioneered a *science of design* and even proposed a curriculum to support it in the universities.

For Simon, anyone who "devises courses of action aimed at changing existing situations into preferred ones" is involved in the act of design. However, this is too broad a statement to be useful for this thesis. Although I have tackled the question of design so far from a broad view, time has come to start directing the efforts towards the specific domain of interest of this work. Thus, in the next sub-section I will begin to converge to the specific domain of education aided by computers, and more particularly on the specific problem of using computers to design teaching and learning experiments, as this is the main interest of this thesis.

2.2.1 Computers, education and design

It is well known that computers have picked the interest of researchers to further investigate both model-centric and process-centric design. In fact, computers have not only joined the design efforts, as other tools did, but have molded the very fashion design is being done in almost every domain of knowledge. And this is not different in the domain of technology enhanced education.

One of the activities in education that take advantage of computers is the design of learning material and systems. However, the name of the particular activity may change according to author and according to date. Historically, this part of the design for learning – which may involve other preoccupations, like designing curricula, for example - has been called *instructional design*. More recently, however, with educational theories like Constructivism going mainstream, and with authors seeing a *constructivist instruction* - at least from a theoretical perspective – as an oxymoron (Jonassen, 1994), some authors are using instead the term *learning design*. There are some distinctions, though, between these two activities, even if the differences may recede in the future:

> The term learning design is associated with a specific field of theory and practice that has evolved from instructional systems design (ISD). ISD is an educational specialism of information design, which has strong leanings towards cognitive and systems science. Its main application has been the design of learning materials and systems: its strengths are particularly evident in distance delivery and self-directed contexts, where it can 'design in' some of the cognitive scaffolding that would otherwise be provided by face to face interactions. Learning design focuses instead on learning activities and on the sequence in which activities are carried out (JISC, 2004).

Since one of the goals of this thesis, is to show that learning design can go well further than simply describing activities and the sequences in which they are carried out, I believe that learning design will sooner or later encompass all activities performed in instructional design. **In fact, one of the main contributions of this thesis is to show that we can apply new software engineering techniques to create programs (one of the tasks of instructional designers) as a transformation of conceptual models (typically created by learning designers).**

Designers are "polyglot" professionals, as they have to "live on the borders between several communities of practice, surrounded by conflicting interests and requirements" (Bodker & Christiansen, 1997). Designers constitute a community of practice in itself, specialized in an independent discipline of design, with their own languages and practices. Therefore, in order to create artifacts for other communities of practice, a designer will have to learn *their* languages and their idiosyncrasies.

The same holds for designers working for the domain of education mediated by technology at large, and more specifically by the computer. They will have to learn a minimum about the vast body of knowledge of Education – which, in this case, is still huge -, before they can apply their skills. Just like we have instructional design and learning design, the name of this particular professional may change accordingly.

As we will see in more details in the coming sections of this chapter, there is a recent tendency among the designers of educational artifacts of clearly demarcating process-centric approaches from entity-centric ones. Now, when it comes to design educational applications, instead of following prescriptive guidelines, designers elaborate metamodels, schemas and design languages. Of course one or more process-centric steps are also required to get to a final product – a software program, for example -, but these will probably be taken over by other designers, closer to the "factory floor" and consequently, more acquainted with the particularities of the product.

To be more precise, this tendency advocates a clear separation in space, in time, in staff, etc. between the entity-centric and the process-centric parts of the design. And this contrasts sharply with the usually long and intricate ISD models -- processes mixing entity-centric and process-centric techniques, as we will see soon -- traditionally used by instructional designers.

Certainly, this affirmation requires a deeper clarification, and I will revisit it later on, after we have seen its theoretical basis. Anyway, always having this distinction in mind, I will use both terms, instructional designer and learning designer, interchangeably.

The *instructional designer* was conceptualized by (J. I. A. Visscher-Voerman, 1999) as a professional who:

- Invents, conceptualizes, or creates concrete products or materials for instructional or educational purposes (ranging from classroom materials to nation-wide school programs).
- Is responsible for the educational, instructional, or pedagogical aspects of the product.
- Is able to reflect on his or her work.

Along the activity of design, instructional designers typically have their minds trespassed by a myriad of doubts, as formulated by (Goodyear, 2005):

> Educational design […] involves several iterations around a cycle of articulating design goals (what am I trying to achieve here?) and educational design commitments (What will I ask the students to do? How will I group them? What reading material will they need? etc.). Within this process, it is common for the designer to make provisional commitments (How do I know what I think until I see what I've designed?) and to backtrack.

Certainly, different approaches will allocate different answers to each of these questions. In general, as explained in the introduction, while entity-centric approaches provide ontological answers for designers, process-centric approaches afford teleological responses. But inside each of these two broader categories, there are several finer grained techniques for approaching design. The following sections will treat, in generic terms, two methods of each category: *design languages* and *models* (entity-centric) and *scenarios* and *design patterns* (process-centric). In the second part of this chapter, these methods will be applied to the specific domain of educational design. In passing, languages and models will be revisited in the Chapter Four, where Domain Specific Languages (DSLs) are discussed in more computational terms.

2.3 Design languages

Languages consist basically of a set of symbols manipulated according to systems of rules, known as grammar. Used for design purposes, they can be called Design Languages and are the most fundamental and emblematic examples of the entity-centric, or structure-centric, design approach.

According to (Rheinfrank & Evenson, 1996) "design languages consist of design elements and principles of composition. Like natural languages, design languages are used for generation (creating things) and interpretation (reading things)."

Gibbons and Rodgers also emphasize the structural nature of languages:

> A design language is a set of abstractions used to give structure, properties, and texture to solutions of design problems. Design languages provide building blocks for designs. Design languages provide categories for thinking about design problems. Design languages provide an important link between technological theory and design practice (Gibbons & Rogers, 2006b).

A priori, there are no fundamental differences between natural languages and design languages – especially if we believe Simon's predication that natural language is "the most artificial [...] of all human constructions". However, a distinguishing feature of design languages, if compared to natural languages, is their reduced scope. That is, they go deep, but not wide. Conversely, natural languages, not being bound to any domain, easily reach a few thousands terms –though this tally significantly differs among people.

On the down side, natural language terms present shallow semantics, overlapping meanings, imprecise definitions, etc. and are highly dependent on the context. Design language terms, in turn, exactly for being constrained to very limited problem spaces, can be very precise, semantically rich, unique and less dependent on the context, sufficient to make them formal. Note that I use the term "formal" here denoting that they can be represented and processed in computers and not that they present semantics, syntaxes, grammars, etc. strictly defined in mathematical terms. Perhaps "semi-formal" would be a better match.

It is a common misconception that design languages are completely independent of context. I personally hold another view and take to heart Rheinfrank and Evenson's words:

> Design languages can be used most effectively when meaning is seen not just as the built-in sense of an object, but also as the quality of *sense making* that objects have and can produce, especially with respect to their surroundings. […] According to this view of meaning, the sense of an object cannot be separated from the experience that the object simultaneously sits in and helps to create (Rheinfrank & Evenson, 1996).

Also, according to them - just like natural languages - design languages are not immutable artifacts, but change as expectations of both designers and users change, evolving through "invention, accident, and other events that create an impetus for transformation".

2.3.1 Creating Design Languages: the Peer Feedback language

What, then, does a design language consist of? Take as an example the text below, a design language describing a teaching strategy (it is in fact an excerpt from a pedagogical pattern – a subject that will be explained later, in Section 2.6). This specific example was taken from the Peer Feedback pedagogical pattern (Bergin, Eckstein, Manns, & Sharp, 2002) and will be used to exemplify the constitution of a design language (Figure 1):

If the **artifacts** were produced in **teams** consider one team member accompanying the artifact as an **agent**. The agent can provide valuable insights for the **review team**. Give each team the chance to report to the whole group on what they have learned, when evaluating the artifact as well as what the agent has learned from the **reviewers**. *If the artifacts were produced by individuals instead*, ask the students to pass the artifact around. *Depending on the size of the whole group, either* again give everybody a chance to report on what they have learned, or assign some time so the artifact **producer** and the reviewer can discuss what they have learned in a **dialog**. You can provide special **feedback** forms to facilitate this.

Figure 1: the peer-feedback language (adapted)

This is a simple, run-of-the mill example, but that serves to illustrate my explanation. Our language above contains **entities** (emphasized here in bold) like artifact, team, agent, feedback, etc., **rules of grammar** (italicized parts) and **actions** performed by the entities (underlined parts). Once elicited, the basic components of our language – called, say, *peer-*

feedback language – could be reused in different situations, just by re-arranging the components differently.

If necessary, we could also represent this language in computers. The specific case of design language used in computer science is called a Domain Specific Language (DSL), a topic that will be treated with a lengthier discussion in Chapter Four, where much of what has been said here about design languages will be restated, though couched in computational terms.

2.4 Models

"All models are wrong but some are useful"

George E.P. Box

Design is never a trivial task. Designers are always dealing with the "new" or, as Schön puts it, with "messy, indeterminate situations" (Schön, 1987). Because "the human reality is too complex for us to be able to deal with it without reductionism" (Meirieu, 1987), designers are permanently attempting to make their task more tangible by reducing complexity while preserving what they consider to be essential.

One way of reducing the complexity of the design task is by constraining the problem space and, to achieve this, designers usually resort to abstractions:

> One of the most important challenges of system design is dealing with complexity. We attack complexity with abstraction. Much of design concerns itself with finding the "right" abstractions in a system, partitioning the system into mind-size, manageable chunks (J. O. Coplien, 1996).

A common way of dealing with abstractions is using models. Models basically allow focusing on the relevant aspects for the design by removing the extraneous parts out of it, or, in other words, by increasing the level of abstraction (Greenfield & Short, 2004) – an idea that can be expressed by the old saying "to separate the wheat from the chaff".

In practice, different sciences have specialized views over the concept of a model, even though two common ideas seem to recurrently appear in almost all definitions, explicitly or implicitly: *simplification* and *purpose*. That is, a model refers to only some aspects of the entity it represents and it does so to serve a purpose. It is in the light of these two observations, that a broad statement like "anything can be a model of anything" (Wartofsky,

1979), from the Philosophy of Science, should be interpreted. That is, anything can be a model of anything if it is simpler –i.e. it abstracts out irrelevant parts of the target system - and if it addresses some purpose - in virtue of which a model is a representation of something else.

Models are of central importance in science, technology and philosophy, as they are simplified representations of things that are too complex, or somehow inaccessible –e.g. too far, too small, too big, etc. - to be studied directly or that simply do not exist yet. So, they can be used to help us learn something new about the things they represent. For example, a blueprint is a model that helps us learn things about the house it represents, without having to be physically in the house (Figure 2).

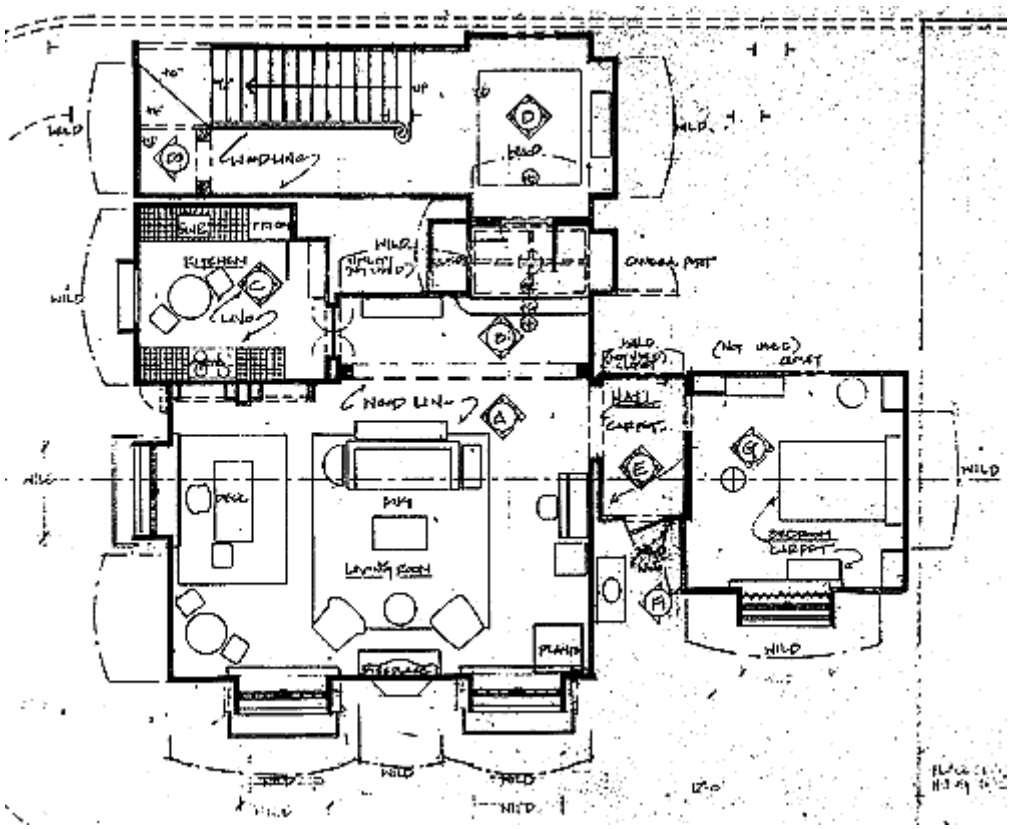


Figure 2: blueprint

(Frigg & Hartmann, 2006) suggests that models perform two representational functions, namely, they can represent a *system* or a *theory*. In the first case, a model is an object that presents an identical or similar structure to the target system and in the latter case, a model is a structure that makes all sentences of the theory to be true.

Models and design are intimately tied to each other. Initially, the theory of design is used to conceive models, and subsequently, it makes use of these models to create the target system. In other words, design theory is involved in both the construction and the manipulation of the model.

Nevertheless, it seems to be an mid- to long term objective, at least in computer science, to do away with this "division", by making both model and designed system "coincide"; i.e. when having a model means having the target system. **In fact, this is one of the central points of this thesis, which suggests the use of model-driven techniques to automatically generate educational applications once educational models are provided**. So, even if model and system are distinct artifacts altogether, from the point of view of the learning designer, they are identical, the latter being just an automatic transformation of the former – transparent to the designer for that matter. Chapter Four will discuss an approach that makes this transformation possible and Chapter Five will describe a prototype that makes use of this approach.

2.4.1 Conceptual models

There are so many types of model –like analytical models, numerical models, mathematical models, phenomenological models, statistical models, iconic models, scale models, toy models, etc.- that, to bring this abundance under control, Frigg et al. (ibidem) suggest to group them according to the answers they provide to some fundamental questions in the realm of semantics ("what is the representational function that models perform?"), ontology ("what kind of things are models?"), epistemology ("how do we learn with models?") and philosophy of science ("how do models relate to theory?" or "what are the implications of a model based approach to science for the debates over scientific realism, reductionism, explanation and laws of nature?"). This thesis is expected to make some contribution to the ontological and epistemological questions.

Concerning the ontological question, the objective of this part of the thesis, however, is not to provide a taxonomy of models and much less to investigate each variation of them. The particular interest of this section is to cover the subject from a theoretical standpoint and just enough to characterize it as an artifact that designers can get a hold of for representation purposes. As for the epistemological question, this thesis proposes a new approach allowing

designers to more easily create expressive models, directly influencing how we can learn about the target system. Also, this section advances a theoretical introduction for the Chapter Four, when I will discuss models from a computational stance.

The essence of the answer to the ontological question can be found on the conceptual models. Fundamentally, *conceptual* (or mental) *models* (Craik, 1943; Gentner & Stevens, 1983; Oakhill & Garnham, 1996) are "representations in the mind of real or imaginary situations"(Johnson-Laird & Byrne, 2000). A more radical approach would be to say that they are the only "windows" we have to natural phenomena. That is, "natural laws", according to this approach, "do not state facts about the world but hold true of entities and processes in the model" (Frigg & Hartmann, 2006).

The importance of conceptual models is that it subsumes, for the intended purpose, all known information about the system it represents. Later on, one may decide which notation – a formula, a drawing, a maquette, symbols, etc – is best adapted to convey this information to other people and even computers, but, *a priori*, notations do not add any new information about the system. For the scope of this thesis, the "things" to be manipulated by designers will be computer-editable notations – e.g. a textual description, a graphical design, a program code, etc. This particular subject will be given a lengthier discussion in Chapter Four, though, when I will explain abstract and concrete syntaxes of Domain Specific Languages (DSLs).

2.5 Scenarios

Contrary to design languages and models, scenarios are process-centric design techniques. The original meaning for the word "scenario", from the Italian, is "that which is pinned to the scenery". It comes from the time when outlines of entrances, exits, and action describing the plot of a play, etc. were literally pinned to the back of the scenery, serving as simple reminders of the plot for the members of the cast[3].

Nowadays, even with its meaning extended to connote the outline of the plot of a play or opera, the distinguishing feature in this broad sense still persists that conveys the idea of scenarios as streamlined descriptions of a sequence of events occurring in a system (e.g., a

3 As described in the *Wikipedia* article about Scenario. Accessed in January, 16 2007.

play, a literary work, a computer program, a learning situation, etc). Roughly, this is the idea that was preserved when this term was imported by other domains, like software engineering, requirements engineering, instructional design, etc.

However, this "overall idea", as shown above, is still too vague to be useful, and if we want to borrow the word "scenario" and make it attractive for our design purposes, we are supposed to come up with a precise enough definition, which will allow us set scenarios apart from other design techniques. Thus, I reproduce three of the most commonly cited definitions of a scenario below, from which we can thus set out to derive the nature of scenarios, by eliciting what makes a scenario to be a scenario:

- "A scenario is a description of an activity, in a narrative form" (B.A Nardi, 1995).
- "A scenario is a sequence of steps describing an interaction between a user and a system" (Fowler, 2004).
- "A scenario is a specific sequence of actions that illustrates behavior" (Booch, Rumbaugh, & Jacobson, 1999).

In all those definitions, the words *description*, *sequence* and *activity* appear, either implicit or explicitly. This is already a good enough beginning. (B.A Nardi, 1995) further suggests that a scenario is not a scenario without, firstly, the inclusion of some *user context*, and, secondly, a *narrative format*, while (Carroll, 1995) posits that scenario descriptions should be detailed enough so that "design implications can be inferred and reasoned about".

I am now able to synthesize a definition that subsumes all main ideas above and that addresses our specific interests about design:

> A scenario is a narrative, i.e., a description of a sequence of events, that includes information on the user context and that is detailed enough to guide design.

As for the word *behavior* mentioned by "los tres amigos", it deserves a comment of its own. In fact, (Harel & Marelly, 2003) describe the **scenarios of behavior** to state the required behavior of systems, during early stages of the development process, as opposed to **state-based modeling,** whereby the complete array of changes of state and outputs are described as a function of inputs:

> This is, in fact, an interesting and subtle duality. On the one hand, we have scenario-based behavioral descriptions, which cut across the boundaries of the components (or objects) of the system, in order to provide coherent and comprehensive descriptions of scenarios of behavior. A sort of **inter-object**, 'one story for all relevant objects' approach. On the other hand, we have state-based behavioral descriptions, which remain within the component, or object, and are based on providing a complete description of the reactivity of each one. A sort of **intra-object**, 'all pieces of stories for one object' approach.

To sum up, any similarity to the entity-centric versus process centric comparison drawn earlier is not a mere coincidence.

2.5.1 Using scenarios

Recalling some of Simon's lessons about the intrinsic purposefulness of the design activity, designers are required to meet user's requirements. For the lack of mathematical rigor when in such situations, they have to make do with techniques that, if do not give them an unequivocal response to their problems, at least provide a rough idea of what to do. Scenarios are such a kind of technique.

"Scenarios put us in another person's shoes" (Verplank, Fulton, Black, & Moggridge, 1993). Telling stories about systems from the users' point of view forces designers to think about the experience of using things *as users*, which might help designers avoid missing vital aspects of problems. On the other hand, partaking these stories ensure that project stakeholders share a sufficiently wide comprehension of the system. Being written in natural language and without technical descriptions, scenarios can be exposed so stakeholders can confirm if they really describe real situations.

Nevertheless, scenarios, alone, are not sufficient as design tools, serving rather to leverage the whole design activity. If we are supposed to completely specify a system, we have to compose them with other more formal representations.

According to (I. Alexander, 2000) scenarios help:

- describe business processes
- clarify system scope
- identify stakeholders, situations, needs

- organize requirements
- guide design and coding
- provide scripts for testing
- improve project communications

To help illustrate the applicability of scenarios, I will provide in the next sub-section an example of their usage for the domain of computer systems design. An example of their use for education will be discussed at length in the second half of this chapter, dedicated to the application of the different design approaches to education.

2.5.2 Scenarios in computer systems design

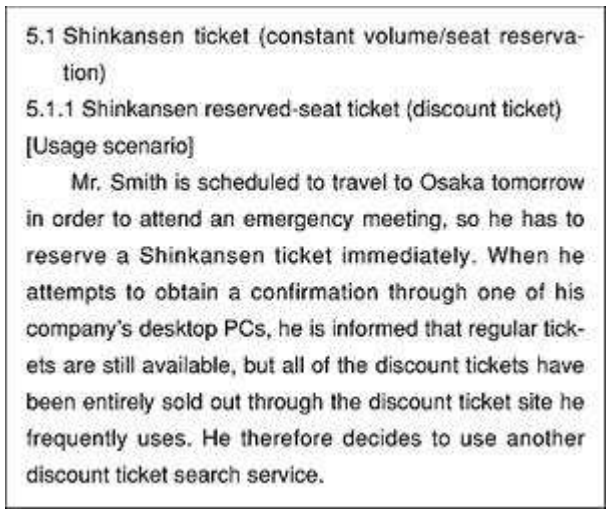
Scenario perspective is regarded as a valuable enhancement of system development practice. As (Carroll, 1995) explains, one thing is to develop technological artifacts to be handled by technicians and quite another is to devise tools to be used by the non-expert, so that they can well adapt to their everyday activities. In the latter case, a designer should also take in account subjective - therefore more difficult to fathom – user requirements.

Contrary to specifying software performance, meeting a user *expectation* is something that cannot be addressed with formal methods. Thus, Carroll goes on to affirm that, as long as we cannot provide "final answers" to questions concerning human activity with enough details so as to guide designers - our best bet will be:

> [T]o develop rich and flexible methods and concepts, to directly incorporate descriptions of potential users and the uses they might make of an envisioned computer system into the design reasoning for that system – ideally by involving the users themselves in the design process (Carroll, 1995).

Thus, scenarios come out as one of these "rich and flexible methods" of design that describe users and the uses they make of a computer system. Utilized as such, scenarios can be applied to close a gap between design and actual use, involving information about the social and environmental setting of a the system "so arguments can be developed about the impact of introducing technology, and the matching between user requirements and task support provided by the system" (Kyng, 1995).

For example, Figure 3 illustrates a typical usage scenario describing facilities offered by a train reservation service and contains valuable information for a designer:



5.1 Shinkansen ticket (constant volume/seat reservation)

5.1.1 Shinkansen reserved-seat ticket (discount ticket)

[Usage scenario]

Mr. Smith is scheduled to travel to Osaka tomorrow in order to attend an emergency meeting, so he has to reserve a Shinkansen ticket immediately. When he attempts to obtain a confirmation through one of his company's desktop PCs, he is informed that regular tickets are still available, but all of the discount tickets have been entirely sold out through the discount ticket site he frequently uses. He therefore decides to use another discount ticket search service.

Figure 3: Scenario (source:(ECOM, 2000))

Thus, from this scenario, designers can apprehend information concerning technical and operational aspects, user needs, issues of concern, etc. In this particular case, this information was used to help them create a web interface.

2.5.3 Scenarios and other design approaches

Scenarios are related to -- and many times confused with -- other techniques used when designing artifacts. Models, use cases and activity diagrams, to name but three, are some of the formalisms that we find convenient to compare here so as to help enhance our understanding of what a scenario is – and what it is not.

Whereas *models,* as a paradigmatic representative of the entity-based design, define, for a given domain, the concepts and the relations between them, *scenarios* -- a typical process-centric design approach -- can be used to describe a specific sequence of how these elements can interact to achieve a goal. To use an example provided by (Dillenbourg, 2003), if someone asks you how to go to a given place, you can provide him an answer either in the form of a scenario or of a model: in the former case, you give him a paper with something like "next street turn right, then turn left at the third semaphore, then..." In the latter case, you just handle him a map and let him decide for himself which path to take.

Another design-aid artifact very popular these days and somewhat related to scenarios is the *use case* (Booch et al., 1999). Scenarios are to use cases as instances are to classes, which means that a scenario is basically one instance of a use case. To (Fowler, 2004), a use case is a "set of scenarios tied together by a common user goal". For example, an "everything-goes-well" scenario of somebody withdrawing money from an ATM could be: insert card, choose operation, type code, type amount, collect card and collect money. But things go wrong as well. What if the code is mistyped? This gives rise to another possible scenario.

An Activity Diagram (Booch et al., 1999) is a flowchart-like diagram distributed along swim lanes, so as to make explicit *who* does what, and as such, may graphically represent the same sequence of events that a scenario would describe through a textual narrative, for example. Like use cases, activity diagrams are part of the Unified Modeling Language (UML).

Although its graphical representation may persuade us of the contrary, *lato sensu,* an activity diagram should be considered a scenario-based notation. Its graphical notation helps us visualize details that would pass unnoticed with plain textual scenarios. They can be further enriched with synchronization bars, forks, joins, guard expressions, etc, making it very precise to describe the flow of activities involved in a process. Figure 4 shows an example of activity diagram:

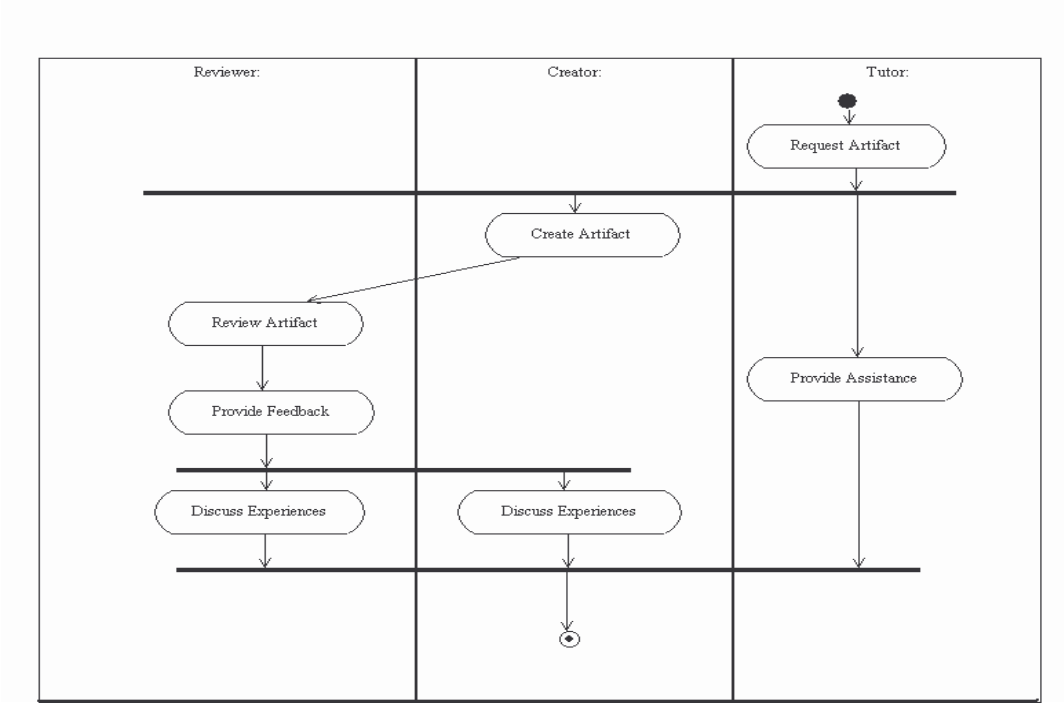


Figure 4: Activity diagram of the Peer feedback pattern.

Thus, this graphical artifact tells us the same "story" that the text of the peer feedback pattern reproduced in Figure 1, only that it highlights elements like activities, role attribution, and time information (sequencing, parallelism, synchronization points, etc.) in a more condensed notation.

2.6 Design Patterns

It was Christopher Alexander, in the book "A Pattern Language: towns, buildings, construction" (C. Alexander, Ishikawa, & Silverstein, 1977) who first proposed the idea of design patterns, providing a language of 253 patterns, for describing buildings, and how they should be designed:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.

However, if Alexander uses the metaphor of rules to better convey the image of patterns, other authors have different interpretations for them, as we will see in the next sub-section.

Design patters are another example of process-centric design approach. This is not to be interpreted as if I was referring to only **process patterns** (Ambler, 1998) and neglecting the so-called **product patterns**, as per classification found in (Conte, Fredj, Hassine, Giraudin, & Rieu, 2002). In my opinion, patterns always describe *series of actions* aimed at producing a final result. That is, **design** patterns, as well as **analysis** patterns (Fowler, 1997), **architectural** patterns (Avgeriou, 2005) or **implementation** patterns (Beck, 2006) describe particular arrangements that should be made to entities – previously defined in an entity-centric design approach or even made "to order" for the pattern – to more wisely attain a goal, which makes of patterns typical process-centric approaches, regardless if their output is a product or a process.

2.6.1 The rationale behind patterns

As explained in the beginning of this chapter, with time, communities develop a knowledge base that is sublimed in the form of a language and of practices. Particularly these practices can be kept in the form of heuristics, guidelines, *ad hoc* procedures, general principles and even as folklore. That is, these high-level and sometimes vague process documentations can

only be mapped into practical solutions by way of sophisticated interpretations, what can be counter-productive. As (J. O. Coplien, 1996) explains:

> There is a cost associated with keeping knowledge locked up as folklore: it makes it difficult to (re)-staff and to maintain legacy systems. Patterns strive to bring such knowledge into the open.

Patterns appear as a solution to this problem, by proposing a way of capturing know-how of experienced professionals, who address recurrent problems in their domain of expertise with insightful solutions. These solutions, which experts develop drawing on their intuition and/or on past experience, are then documented in nearly standardized forms, enabling their reuse. That is, patterns solutions are described in a way that is sufficiently abstract, allowing their reuse, but concrete enough to enable solving real problems. Furthermore, these solutions have a name, which enable them to enter the community's language. The collection of patterns form then a pattern language, as it will be discussed in section 2.6.4.

To summarize, patterns are a way of gaining experience on a given domain without having to spend years on it, passing through all the same situations that other have already faced.

2.6.2 Other metaphors for patterns

Just like with design, we can better grasp the meaning of patterns by making analogies with other, more familiar things. Patterns can, for example, be likened to **micro-architectures**. The idea here is that the particular problem tackled by a pattern is usually much smaller than the whole problem; that is, a single pattern will contribute only partially for the complete design. This is very clear in the work of Alexander, when he proposes, for example, a pattern that suggests suitable locations for street windows. In this case, this pattern will solve only a small part of a much bigger problem, which is the design of a house. Also in the area of software design this metaphor for patterns is present:

> [Design patterns] are micro-architectures: structures larger than objects but not large enough to be system-level organizing principles. (J. O. Coplien, 1996)

Patterns can also be seen as **vehicles for encapsulating knowledge**, in the sense that "pattern names are abstractions that encapsulate implementation knowledge, creating a new and more powerful language" (Greenfield & Short, 2004). Some people prefer to think of patterns as

tools, which "explicitly focus on context and tell the designer when, how and why the solution can be applied" (Welie, Gerrit, & Eliens, 2000).

One of the most widely recognized images for patterns is that of **best practices.** That is, they can be viewed as compilations of particularly effective ways of doing things. Being pieces of literature that can be consulted by beginners, patterns enable them to stand on the "shoulders of giants". As a consequence, the success of patterns hinges a lot on the confidence beginners place in their author(s), the safest guarantee that they really represent *best practices* – and not just practices.

Particularly in computing, patterns can additionally be seen as explicit **programming construct**s, based on "the ability to codify patterns as generic software components" (Andrews, 2002).

2.6.3 Pattern forms

Design patterns are expressed in forms. Different authors craft forms containing sections that best adapt to their intent, which gives rise to an abundance of notations that may significantly differ from one another, both syntactically and semantically. Some of these forms are more widespread than the others, for historical reasons.

The Alexandrian form is considered the original pattern form. Syntactically, it is very loose and offer no significant help if someone ever feels tempted to "automate" them. Barring a "Therefore" introducing the solution section, which can be used to identify the beginning of the *solutions* section, the transitions between different sections are mostly made through visual cues (diamond-shaped symbols, bold-typed and italicized fonts, etc.), constituting then a "nightmare" for parsers. Figure 5 shows an Alexandrian form.

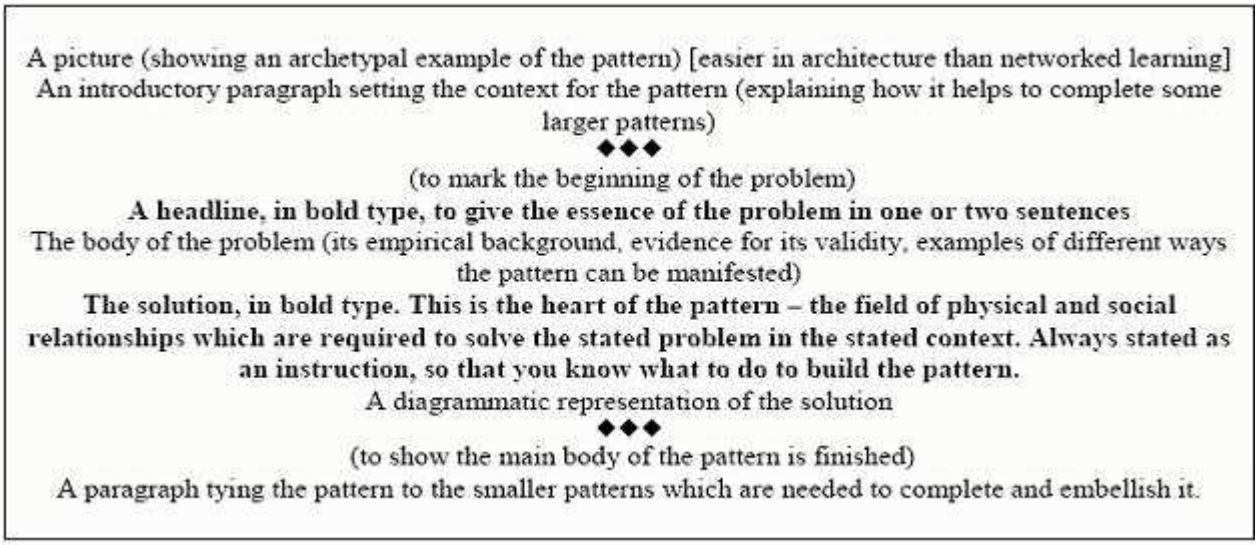


Figure 5: Alexandrian Form

Another well known form is the one used in the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Erich Gamma, Helm, Johnson, & Vlissides, 1995), which introduced the notion patterns for the Computer Science community. It is called the GOF ("Gang of Four") form and presents the following sections: Pattern Name and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses and Related Patterns.

Regardless of the specific form, though, they all seem to respect a "meta-form" composed of *metadata* sections, *core* sections and *optional* sections. Usually placed at the beginning, we have the metadata-type elements -- like *author*, *date*, *version*, *also known as* --, important to label the pattern. Then we have the four core elements -- *title, problem, context* and *solution* -- which alone account for the real identity of the pattern. Take one of them out and you won't have a pattern, change one of them and you'll have a different pattern. Lastly, we have the optional elements, which add features and grace to a pattern, making it more expressive and useful: forces, examples, *related patterns, contra-indications, resources, consequences* are some of them. Optional sections significantly change according to the adopted form.

2.6.4 Pattern languages

Whereas patterns can be a good solution for fine grained problems, linking related patterns can be useful to address more complex processes in a step-by-step approach. To tackle this

issue, Alexander proposed the concept of a Pattern Language, which consists in a set of patterns working together.

Interesting enough, Alexander has anticipated, from the very outset, the way how patterns could become a part of the communities' *language*. More than just being best-practices, patterns are best-practices with a name. In this respect, the title of the pattern can act as a handle and make part of the vocabulary that practitioners and designers can use to easily express sometimes complex strategies, practices, etc. One handle may be worth a thousand words, facilitating the communication among professionals.

This relation between patterns and languages has been brought out by (Greenfield & Short, 2004):

> Any language defines a set of abstractions, which form its vocabulary, and a set of rules, which form its grammar, for combining those abstractions to create expressions. Both the vocabulary and the grammar of a pattern language can be expressed as patterns.

Various authors proposed categorizations of the relations between different patterns of the same language:

- *Generalization/Specialization, Requires as predecessor/has as successor, Can be supported by* and *Can be implemented by* (Voelter, 2000);
- *Uses*, *refines* and *conflicts* (Buhler, Starr, & Weichhart, 2005);
- *Vertical relationships* (hierarchical) and *Horizontal relationship* (semantic) (Botturi, 2003);
- *Requires, Refines, Uses* and *Alternative* (Conte et al., 2002).

2.6.5 Writing patterns

Writing good patterns is *very* difficult. (Brad Appleton)

Pattern design is more often than not a collaborative activity. Using *wikis* as a collaborative infrastructure, patterns are developed in an iterative work that goes through submissions, discussions and revisions. For the case of the computing area, there are even annual workshops organized in international conferences, called Pattern Languages of Programs (or

simply, PLoPs), where pattern writers get together to discuss proposals for patterns and to have their patterns reviewed by fellow authors.

Patterns can be generated in two distinct manners: by *induction* and by *deduction*. In the first case, the pattern creator starts from a specific problem and, recognizing that the solution provided can be generalized to other related situations, decides to formulate a pattern, spelling out how this could be done. Conversely, a deductive approach is used when the pattern author, starting from a more general principle, conceives one or more design patterns, in an effort to reap more practical and immediate results.

The E-LEN project (E-LEN, 2005) adopts this same paired classification, though using instead the terms *opportunistic* approach –for the inductive method – and *top-down* approach – for the deductive mode. (Baggetun, Rusman, & Poggi, 2004) have scrutinized this classification further, proposing subdivisions for each of these two general modes. For example, the inductive mode is broken down into: from instance to pattern; from observation of human behavior to pattern, etc; and the deductive mode can be practiced by particularizing from metaphors to patterns, from mindmaps to patterns and from experience to patterns.

Ultimately, the objective of those who work with patterns is to create a pattern language, composed of several related patterns addressed to a particular domain. To this purpose, (Cunningham, 1994) proposed some practical tips for writing down pattern languages, which, even if described in a very casual manner, should be of use for pattern writers.

2.6.6 How patterns can be used in design

As we have seen earlier in this chapter, in order to be useful for design purposes, the different representational approaches must somehow scale down the complexity of real systems. In this section, we are particularly interested in the idea of design *as patterns*, i.e., how *design patterns* help us deal with complexity and a clue to this can be found in the seminal book that vastly improved the discipline of object-oriented software design:

> Once you know the pattern, a lot of design decisions follow automatically. (Erich Gamma et al., 1995)

That is, once we know which pattern to apply, the solution space gets significantly constrained by the solution provided by the pattern. So, the problem comes down to

identifying when a pattern is applicable for a given problem. This identification is accomplished by recognizing one's own problem as the one described in the *problem* section of a pattern form, and ascertaining that the contexts of the real problem and the one described in the *context* section also match.

The actual application of the pattern boils down to designing a solution to the real problem in the light of the *solution* section of the form. More specifically, the application of the pattern is achieved through imitation: first a duplication of the pattern solution, followed by an adaptation of it to the actual context (Rieu, Giraudin, Saint-Marcel, & Front-Conte, 1999). Admittedly, this is a very rough description, compared to real life situations, in which further complication will likely impose, but an approximation good enough at this introductory level all the same.

I would like to terminate this section with a caveat, to summon the attention to a disadvantage we may find in the use of patterns:

> Design languages usually evolve gradually. They become a deeply held tradition, are difficult to change, and are even more rarely questioned. People tend to assume that they are valid and to continue to work through them, rather than to think about them and their appropriateness. (Rheinfrank & Evenson, 1996)

That is, a pattern language – as any design language - should not stiffen attitudes and novel solutions should always be considered; since "today's original ideas will be the patterns of the next decade, it's important to keep innovative developments alive" (J. O. Coplien, 1996).

This section wraps up the generic discussion about design. The remainder of this chapter will discuss different design approaches applied to education that will be important for the ensuing chapters.

2.7 Pedagogical Scenario

To stimulate collaboration, problem solving and learning, access to learning material, etc. teachers can make use of pre-built sequences of situations that will describe - and guide - the unfolding of events. This description is what we call here a *pedagogical scenario*.

If in the domain of computing it is not easy to find a generally accepted definition for *scenario*, in the domain of learning design this task, surprisingly enough, is a little bit smoother. There has been a convergence of what a scenario is in this domain – at least much more consensual than in computer systems -, even if the word *scenario* is not necessarily present. Browsing the book (Rob Koper & Tattersall, 2005), for example, we find expressions like pedagogical *design,* pedagogical *method*, pedagogical *approach* and pedagogical *technique*, all of them referring to about the same idea of a *pedagogical* or *learning* scenario.

To illustrate this convergence, I reproduce three definitions for a Learning Scenario:

- "[A] a sequence of phases within which students have tasks to do and specific roles to play" (Schneider, 2004).
- "The subset of the scenario grouping activities performed by a learner and their associated resources" (Paquette, 2004)[4].
- "[T]he description of the progression of a learning situation or of a unity of learning in terms of roles, activities and environments, but also in terms of the knowledge involved (Pernin & Lejeune, 2004).

According to these definitions, pedagogical scenarios preserve the basic characteristics of scenarios as seen in section 2.5 – i.e., narrative and sequential – and add to the concept of *activity,* already mentioned, the new concepts of *role* and manipulated *resources*. In so doing, they echo Koper and Tattersall's opinions of a learning design:

> The key principle in learning design is that it represents the learning activities and the support activities that are performed by different persons (learners, teachers) in the context of a unit of learning (Rob Koper & Tattersall, 2005).

Another definition for a pedagogical scenario, which departs a little bit from the "pattern" found above, is reproduced below, for comparison's sake:

> [A] general approach that guides the activities of a group of learners towards the achievement of certain learning goals. A scenario involves not only learning concepts but also operational strategies. It reflects the basic set of values, beliefs, goals, and assumptions of the conceptual models that are

[4] Paquette also considers subsets grouping activities performed by all the other actors and their associated resources, which are called the Assistance Scenario.

> "translated" into empirical models and practical guidelines (van Diggelen, Overdijk, & De-Groot, 2005).

At this moment, it seems opportune to pass a remark about the convenience –of lack thereof- of using the word *scenario*. In fact I have been using it here more for legacy reasons than for the particular appropriateness of the term. Since scenarios convey an indissociable temporal component, I'm afraid it is not always applicable to cases when what we really mean is the more generic concept of a *learn facilitating experiment*, which is a much wider idea than the one conveyed by the metaphor of a scenario.

Not every teaching/learning situation requires *time* to be a first-class element. What I mean is that, even if it is always present, it does not necessarily have to be explicit, as scenarios implicitly force us to think. For example, the description of whole learning experiences can be made in terms of timeless elements, like motivations, attitudes, reasons why activities were done – or not done -, values, beliefs, goals, in summary, elements that make sense not only of events and actors involved, but also of the context in general (geographic, social, economic, etc.) and that are rather difficult to plot on a time-line.

To address the shortcomings of strictly sequential descriptions, as we will see in the next section, the IMS-LD specification – to be discussed in Section 2.9.1 - proposes a workflow model to its scenarios. However, even if it does enrich the control logic, by proposing concurrent flows, conditional loops, jumps, etc., it does not escape the temporal element – in fact, it reinforces.

Constructivist designers usually limit themselves to providing rich environments where meaning can be negotiated, and where ways of understanding can emerge (Hannafin, Hannafin, Land, & Oliver, 1997). In this case, there is not necessarily a *plot* guiding what must be done to achieve a goal, much less as a function of time.

Does this mean that, since we cannot impose time constraints, we cannot impose constraints at all and, by definition, we cannot do design? Or else, does this mean that designers are hand-tied and scenarios are useless? I don't think so. Taking the simple feedback language given as an example of a design language in section 2.3, where we had the concepts of "team", "feedback", "artifact", etc., the "feedback" concept does impose a constraint on the experiment, by specifying that learners will have to produce a critique on the artifacts of their

peers. It is not important, though, to state *when* this feedback will be produced, or even its order in relation to other activities – it is not difficult to picture real spontaneous situations in which students ask and/or provide feedback *during* or even *before* the creative process. That is, we don't need a time regulated model to specify when the feedback should be made. In such case, a scenario that hard-codes the common sense solution –i.e. feedback comes *after* the conception of the artifact – could cause more harm than good.

Certainly, when those constructivist environments are "executed", arrangements of activities will naturally ensue as a consequence of a "search for authentic tasks". Similarly, specific objectives will emerge, on-the-fly, that are "appropriate to the individual learner in solving the real-world task" (Bednar, Cunningham, Duffy, & Perry, 1992). But these arrangements will be spontaneous, improvised sequences that might hardly match most people's expectations for a narrative – and that therefore would be unlikely to be designed a priori.

All that being said, I will keep on using the expression *learning scenario* in this thesis, mainly for the lack of a better substitute. However, it should preferably be interpreted not according to the classical definition of a scenario, but as any formal or informal description, a priori or a posteriori, of *teaching-learning experiments*, including their context, aimed at documenting it and/or reproducing it in different contexts.

2.8 Instructional System Design

Having devoted considerable space in this chapter to various questions concerning the design of learning scenarios, in a rather conceptual level, this section now is dedicated to a somewhat different body of theory, concerned with how systems should be designed in order to leverage – rather than hamper – the theories, models, scenarios, etc. previously developed by the experts in education. This body of theory is called Instructional Systems Design (ISD).

As explained in the section 2.2.1, where I discussed the origins of the expression *instruction designer*, as contrasted with the newer *learning designer*, ISD proposes a process-centric approach that describes steps to be followed by designers in order to develop applications, taking account of both design and educational concerns. Schematically, this could be diagrammatically represented as in Figure 6:

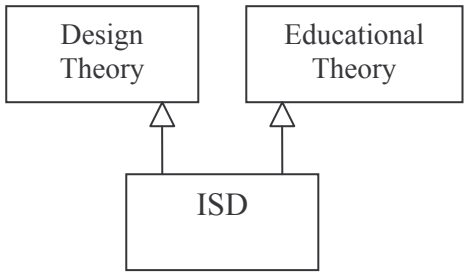


Figure 6: ISD and its ascendants

However, this separation is not always clear, as (Gibbons & Rogers, 2007) explicates:

> [ISD] often includes an admixture of design processes with instructional theory, so that the design process appears to be theory-derived. The result has been a set of loosely-specified, non-standard, highly-variable design activities held up professionally more as an ideal than as a criterion, and that conflate the design process with specific domain theories of instruction.

As for the left side of the inheritance above, in the traditional ISD, this particular design theory is clearly based on the **Process Design Theory** (Rolland, 1993) and, more specifically, on the Software Engineering discipline (Sommerville, 1989). That is, while Software Engineering deals with processes for structuring the design of software in general, ISD is a particularization of it for the design of the technological artifacts -- more notably, computer-based -- used in education.

The Process Design Theory, and consequently Software Engineering and ISD, is organized around *process models.* The word *model* here, nevertheless, presents a somewhat different connotation from the one explained in section 2.4, and should be interpreted rather as a something idealized, worthy to be imitated. A more formal definition of a process model is to be found in (Rolland, ibidem):

> A *process model* is a description of process. Most often a process model is used for a descriptive purpose i.e. to describe "how things must/should/could/be done" in contrast to the process itself which is really what happens.

According to the above definition, then, a process is an instance of a process model, which, in turn, is an instantiation of a process metamodel. This theory is depicted in Figure 7.

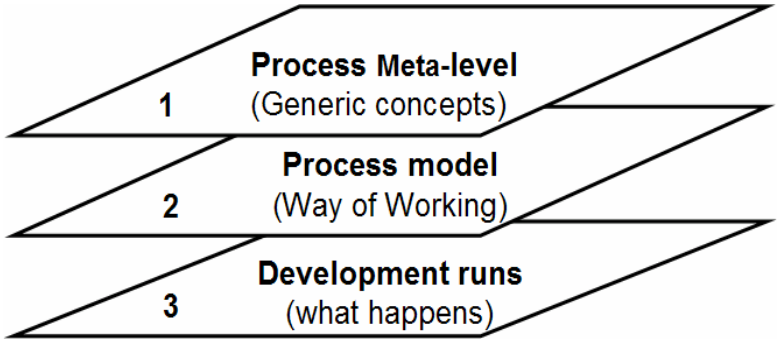


Figure 7: Abstraction level for processes. Source: (Rolland, 1993).

I am not interested here, though, in exploring the meta-levels of this theory, my only concern being to clarify the role of process models in the more traditional instructional design. In practice, ISD process models are realized as *guidelines* for building effective training and performance support tools and, just like more conventional software engineering, prescribes a partitioning of the whole process into a number of stages.

Perhaps the most common process model used in ISD is the ADDIE Model. ADDIE is an acronym referring to the major steps comprised in a generic ISD process: Analysis, Design, Development, Implementation, and Evaluation. Though widely used as an umbrella term for all ISD models, the original source of the acronym is unknown (Molenda, 2003). The ADDIE model is not necessarily a sequential process, as illustrated below:

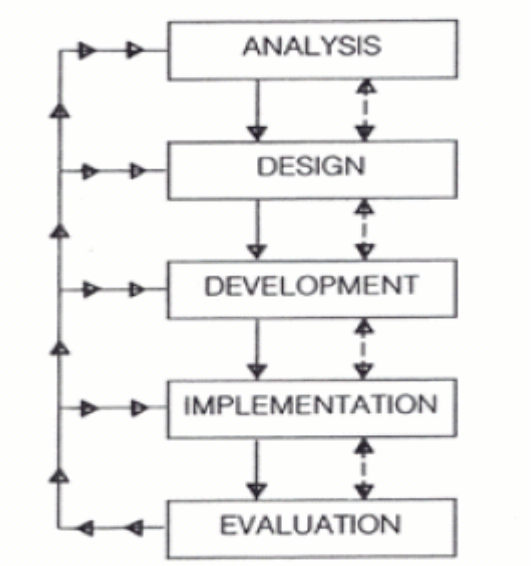


Figure 8: the ADDIE Model (source:(Grafinger, 1988))

Using the ADDIE model, designers are supposed to cycle through stages in which they analyze learner characteristics, task to be learned, etc.; develop learning objectives, choose instructional approaches; create instructional applications and material; deliver or distribute the instructional materials; make sure the materials achieved the desired goals. However:

> These methods are purported to be derived from general systems theory, but the methods are often taught with a high degree of local variation without much reference to the foundational theory (Gibbons & Rogers, 2007).

As for the educational side, (the right side inheritance of Figure 6), traditional ISD draws, not directly on generic Educational Theories, but on the more pragmatic **Instructional Theories**, which reflect "a particular theorist's view of effective *instructional* structures and operations during instruction" (Gibbons & Rogers, 2007).

Much of the foundation of the field of Instructional Theory is based on the works of (Bloom, 1956), (Gagne, 1985), (Reigeluth, 1983) and a few others. Instructional Theory is traditionally rooted in cognitive and behavioral psychology, which advocates acquisition of knowledge and skills as the goal of education. Figure 9 shows the expanded ISD lineage.

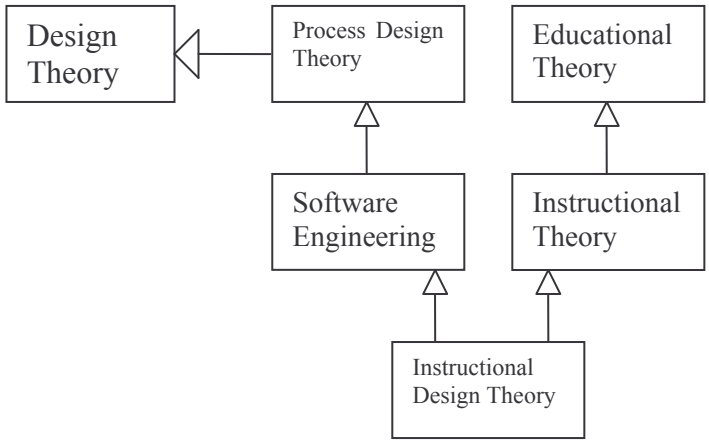


Figure 9 : ISD and its ascendants (expanded)

As explained before, new technologies are molding a different professional altogether, significantly changing the way they work. If ISD is a primarily process-centric approach, next section I will address EMLs, with their primacy on the entity-centric perspective.

2.9 Educational Modeling Languages and IMS-LD

There is a rich diversity of teaching and learning strategies used in real educational situations and Educational Modeling Languages (EMLs) are design languages tailored to represent these strategies. Unfortunately, the EML acronym has been used in the literature to describe different languages with completely distinct intents.

A survey (Rawlings, Rosmalen, Koper, Rodriguez-artacho, & Lefrere, 2002), for example, enumerates a few design languages (or, more specifically, DSLs, as we will see in Chapter Four) under the same "EML banner" when they have very little, if anything, in common with one another. For instance, whereas languages like CDF (CDF, 2003) or LMML (Süß & Freitag, 2002) have been conceived for material aggregation and/or knowledge management purposes, others like IMS-LD (IMS-LD, 2003) and MISA (Paquette, 2004) – more specifically MISA's instructional model - are mainly used to represent the instructor's teaching approach.

Since in this thesis I am only interested in the pedagogical approach of a learning scenario, I am going to focus solely on EMLs connoting pedagogical strategies. Furthermore, other design languages exist, which are not included in this survey, but which also are typical representatives of the pedagogy-centered EML, like the Learning Design Language - LDL (Ferraris, Lejeune, Vignollet, & David, 2005) and the Cooperative Problem-based learning Metamodel – CPM (Laforcade, 2004) and many others.

Much of the recent research dedicated to improve ways of designing and developing e-learning applications has been centered on standard design languages. As (Friesen, 2004) explains to us, in e-learning, standards are often multi-part assets, typically consisting of a "conceptual" model, one or more "bindings", which specify how the conceptual model is expressed in a formal idiom -- most often XML -- and last, an Application Programming Interface or "service definition", which defines standardized "contracts" between cooperating systems.

This means that at the heart of those applications are conceptual models that describe key entities involved in e-learning systems. This can be considered as another attempt to "recover forward momentum with regard to instructional theory and its application to instructional

designs, given that process models of design are proving insufficient and are aging" (Gibbons & Rogers, 2006a).

Examples of these standards are learning object metadata (LOM, 2002), e-portfolios (IMS-EP, 2005), learner information (IMS-LI, 2005), and of special interest for this thesis, the Learning Design, or IMS-LD, which serves to describe pedagogical scenarios, as we will see in the next sub-section. Generically speaking, as already stated, a design language serving to represent pedagogical scenarios is called an Educational Modeling Languages, and IMS-LD is just an example of it. Please note that, in this thesis, the expressions "modeling language" and "design language" are used interchangeably.

The survey (Rawlings et al., 2002) mentioned earlier, published by the CEN/ISSS Workshop on Learning Technologies, provides the following definition for an EML:

> An EML is a semantic information model and binding, describing the content and process within a 'unit of learning' from a pedagogical perspective in order to support reuse and interoperability.

(Rob Koper & Tattersall, 2005) lists several requirements for an educational modeling language – which they mistakenly attributes to the language's *notation* - , providing the basis for the Open University of the Netherlands' modeling language, not surprisingly called Educational Modeling Language (EML, 2000). These requirements are reproduced below:

1. The notation must be comprehensive. It must describe the teaching-learning activities of a course in detail and include references to the learning objects and services needed to perform the activities.
2. The notation must support mixed mode (blended learning) as well as pure online learning.
3. The notation must be sufficiently flexible to describe learning designs based on all kinds of theories; it must avoid biasing designs towards any specific pedagogical approach.
4. The notation must be able to describe conditions within a learning design that can be used to tailor the learning design to suit specific persons or specific circumstances.
5. The notation must make it possible to identify, isolate, de-contextualize and exchange useful parts of a learning design (e.g. a pattern) so as to stimulate their reuse in other contexts.
6. The notation must be standardized and in line with other standard notations.
7. The notation must provide a formal language for learning designs that can be processed automatically.

8. The specification must enable a learning design to be abstracted in such a way that repeated execution, in different settings and with different

Contrasting with ISD approaches, which typically require a design team to participate in every stage of the development of educational applications, EMLs separate the development effort in two distinct phases, which can be performed by two different types of designers, with possibly different skills:

- "Upstream" designers: conceive the EML itself, which are realized as the multi-part assets described up above, consisting of the conceptual model, the bindings (i.e. formal notations) and the APIs – sometimes exposed in the form of a software framework so as to facilitate their utilization.
- "Downstream" designers: work closer to educational practitioners, their job being to create instances of the conceptual models representing practical learning situations. Their task can be significantly smoothed out if upstream designers provide the required tooling, such as:
  - editor(s) for the conceptual model, which will permit to more easily generate a "binding" -- usually XML -- for their mental design;
  - a software framework, which will certainly include a runtime engine, serving to read and interpret the bindings generated by the editor.

Concerning the final learning scenario, upstream designers perform a primarily *entity-centric* design, by creating the elements (concepts, properties, rules, etc.) that define the scenario's ontological universe while downstream designers conceive different ways of *how* these elements can be combined so as to meet an immediate goal, in an eminently *process-centric* approach.

Contrasting with ISD designers, EML designers (that is, the "upstream" designers) no longer offer turn-key programs to the final users, but configurable frameworks that will be used by the "downstream" designers to actually produce and deliver the final product to the user. This separation has clearly influenced the way designers do their jobs. As I mentioned earlier in this chapter, this even justifies a change in the designation of the professional – from *instructional designer* to *learning designer*.

In Chapter Three, I will discuss yet another approach, called Multi-EML, which is one of the contributions of this thesis. Multi-EML's intention is to ease the process of creation of metamodels (i.e. design languages), making it accessible to downstream designers. That is, while with the plain EML approach, upstream designers conceive fixed metamodels, leaving to downstream designers the only possibility of arranging their concepts in a model, with the Multi-EML approach, both model and metamodel are configurable. By doing so, the Multi-EML provides a much more flexible and rich environment, facilitating the task of educational metamodel creation for both upstream and downstream designers – or else doing away with this hierarchy altogether.

2.9.1 IMS-LD

Recently, EML communities have directed their attention towards the conception of a single standard EML, powerful enough to represent every possible learning scenario. One of the first serious moves in this direction was the adoption in 2003 by the IMS Global Learning Consortium (IMS) of the OUNL's Educational Modeling Language, giving rise to the Learning Design specification[5]. The IMS Learning Design (LD) Specification (IMS-LD, 2003) was proposed as a standard capable of supporting the modeling of units of learning in accordance with different pedagogical approaches, that is, as *the* standard EML.

IMS-LD was brought -- more or less mid-stream -- into an IMS working group that was originally charged with the task of developing a specification related to instructional design (Friesen, 2004). It appeared as a response to a moment in which the e-learning community had its attention and efforts polarized towards the development of a standard that allowed representing *what* to learn (i.e. about *content*).

Presently, most online training is organized around Learning Management Systems (LMS), which provide a means for building courses, managing roles and groups and building in the various 'services' (e-mail, conferencing, chat, etc.). However, the course structuring functionality in LMSs typically means structuring content - they argue. To IMS-LD sponsors, the activity management capabilities in LMSs to date have been very limited. And IMS-LD

---

[5] Specifications represent standards early in their development, prior to receiving approval from standards bodies, often being, as a consequence, experimental, incomplete and rapidly evolving. Specifications contrast with *standards*, which are final regulatory documents that evolve from specifications.

proposes "moving toward being more concerned with runtime issues, interactions and activities" (Blinco, Mason, McLean, & Wilson, 2004). As (R. Koper, 2002) puts it:

> A lot of learning does not come from knowledge resources at all, but stems from the activities of learners solving problems, interacting with real devices, interacting in their social and work situation.

Using the classification created in the introduction of this chapter, IMS-LD too is composed of an entity-centric part, where its concepts are defined, and of a process-centric part, where these concepts will be arranged, aiming at a particular scenario.

As for the structural part, IMS-LD vocabulary reflects an *activity-centred*[6] approach (Scheunpflug, 2001, *apud* (Allert, 2004)) and evolve around concepts like activity, role, (learning) objective, environment, etc. The complete tree of concepts used in IMS-LD is reproduced in Figure 10 - adapted from (Olivier & Tattersall, 2005).

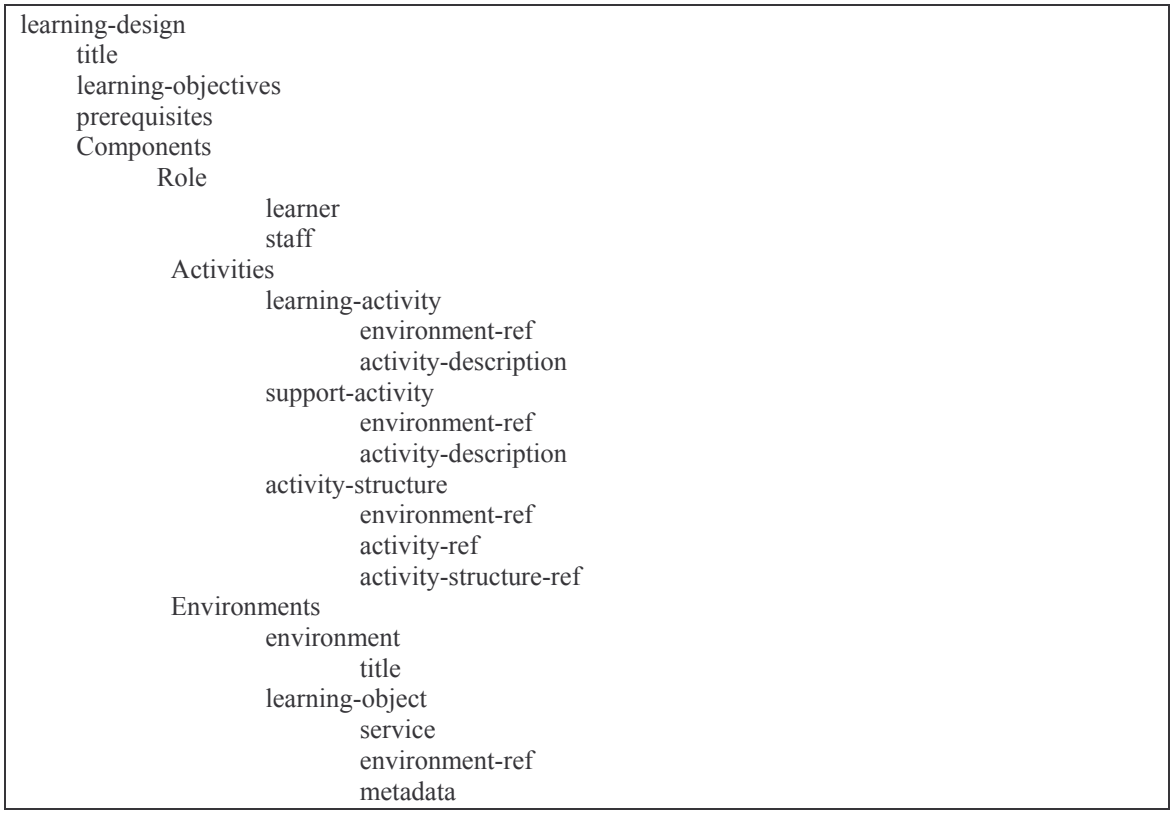


Figure 10: IMS-LD structural elements

[6] The activity centred approach should not to be confused with the Activity Theory (B. A. Nardi, 1996), (Engeström, Miettinen, & Punamäki-Gitai, 1999).

The environments are composed of learning objects (e.g. text files, PDF files, video, etc) and services (send-mail, conference, monitor and index search).

As for the process-part, it actually describes the specific pedagogical *scenario*, i.e., it defines the order how components will be combined. To facilitate its apprehension by learning designers, it uses the metaphor of a theatrical play, as illustrated in Figure 11.

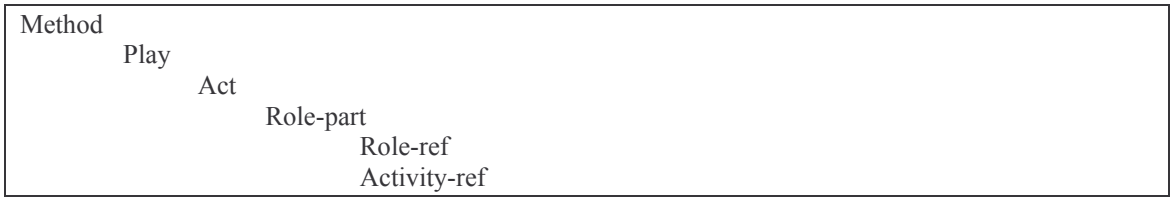


Figure 11: IMS-LD process elements

That is, these elements reflect a scenario, allowing to set up a "play", which arranges how activities take place. Each element described above is linked to the subjacent one according to a "has-a" relation, such that a Method has one or more Play, which, in turn, contains one or more Act, and so on. Once they are all put together, the learning-design is ready to be uploaded to a runtime engine, which will execute the Plays in parallel, Acts in sequence and, within Acts, Role-parts, also in parallel. More sophisticated dynamics can be obtained with conditional logic, though. Note that the structural components of the scenario (roles, activities and environments) are referenced by the Role-part elements, which can be better seen in Figure 12.

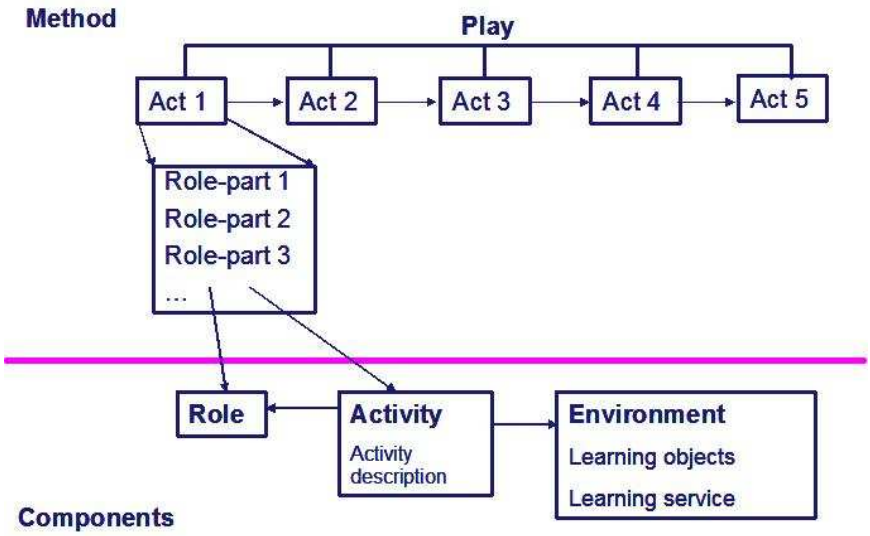


Figure 12 : Relating structure and process in IMS-LD (source: (Olivier & Tattersall, 2005))

This aspect of the IMS-LD reminds to some extent the Model View Controller (MVC) design pattern[2], being the *model* part of MVC represented by IMS-LD structural elements (roles, activities, properties, environments and their sub-elements) and the *controller* represented by the IMS-LD "method" element (with its sub-elements). It is the controller that connects the "model" elements together according to a sequence, as illustrated in Figure 12. The MVC's *view* part is not prescribed by the specification, being open to each "player" implementation.

Physically, a learning scenario is an archive file containing all files required by the runtime engine to execute the scenario. This includes a manifest file – an XML file describing the scenario itself - and all files referred to by the manifest (i.e. the actual contents used by learners and instructors). This archive file is called a **Unity of Learning** (UoL) and respects the IMS Content Package specification (IMS-EP, 2005), for packaging learning materials into interoperable, distributable packages. Figure 13 exhibits the structures of an IMS generic content package and of a Unity of Learning.

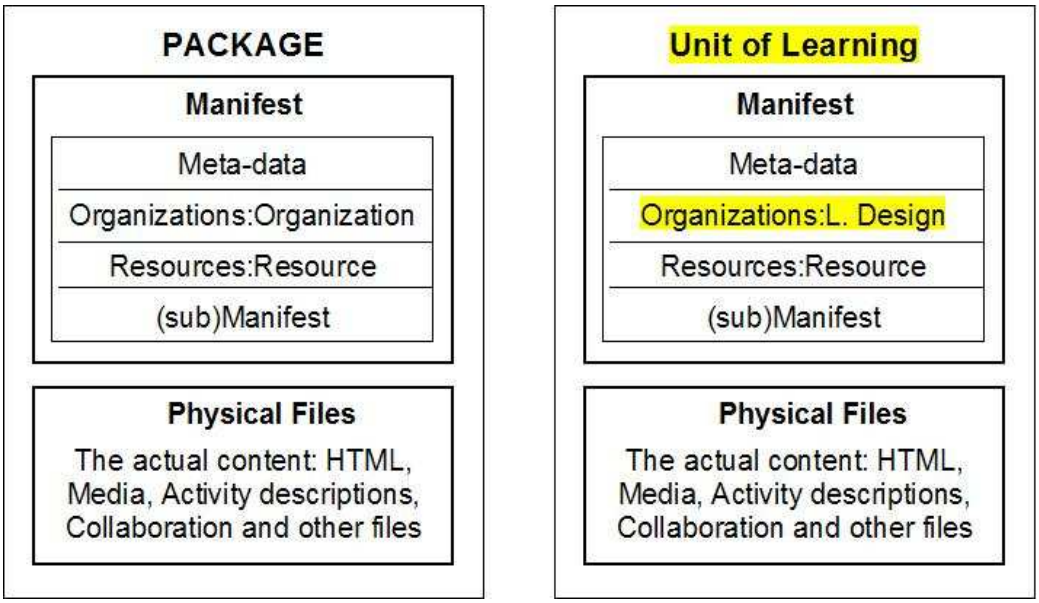


Figure 13: IMS-CP (left) and UoL (right)

We should also consider as part of the IMS-LD, the CopperCore (Martens & Vogten, 2005), a reference implementation for an IMS-LD runtime engine, allowing UoLs to be "executed", and also many other tools, for editing, delivering, playing, etc. UoLs, some of which are described in (Griffiths, Blat, Garcia, Vogten, & Kwong, 2005).

As a final remark, I believe that building effective pedagogical scenarios is one of the challenges for the future of e-learning, since it will allow to "model" educational situations and implement them in virtual settings. In that case, creating a repository of useful educational approaches will be a necessary - but non trivial - task. As e-learning tools begin to make use of learning scenarios, users will look for generic templates embodying certain pedagogical techniques, in the hope that they can then be adapted to meet specific requirements. IMS-LD has been tailored to offer a solution for this problem. However it is not the only solution and certainly not the best for many scenarios, especially those drawing on the constructivist theory of learning. An alternative to IMS-LD is described in the next chapter.

2.10 Pedagogical Patterns

Originally proposed by (C. Alexander et al., 1977), design patterns are nowadays a cross-cutting concept, adopted by several domains. If the books in which Alexander laid down the principles of design patterns date back to the seventies, only in the last ten years or so has the scope of the pattern expanded to include domains as diverse as group work, software design, human computer interaction, education, etc. *Pedagogical Patterns* are targeted at education; they seek to find best practices of teaching. According to (Bergin, 2001):

> The intent [of pedagogical patterns] is to capture the essence of the practice in a compact form that can be easily communicated to those who need the knowledge.

Although widespread in the software industry, the use of design patterns is still only nascent in the educational field. Whereas software developers make regular use of mature patterns, the educational community as a whole is still far from including pedagogical patterns in their everyday tool-box.

Figure 14 reproduces an example of a pedagogical pattern (called "One Grade for All") taken from the "Feedback Patterns" pattern language (Bergin et al., 2002). Here we see that it stands by the Alexandrian form, as shown in Figure 5.

ONE GRADE FOR ALL *

This pattern was written by Jeanine Meyer as *Assigning and Grading (short) Team Projects* [JM] and revised by Helen Sharp.

You have a short team project to mark, one that has lasted between one class session and three weeks, and does not form a substantial part of the overall course marks. You may have assigned the teams through Teacher selects Teams [EBS].

❋❋❋

You want all team members to benefit equally from the teamwork experience, in terms of grading and learning outcomes. But some team members often put in more work than others.

Teamwork can deepen the learning of subject matter because the projects can be more substantial, but the benefits will only accrue to the individuals if each of them puts in an equal amount of effort.

❋❋❋

Therefore, grade the team's work based on a presentation, which may be given by any one member of the team. Choose the presenter on the day so that each member of the team has an equal stake in preparing for the presentation and will have prepared equally. Give each team member the same grade. This resembles the real-world situation. You can tell teams to divide up the tasks in any way they think is appropriate, but that everyone must understand everything. Students will take responsibility for the inter-personal issues and project management issues. Students learn from each other. You may find that individual students demonstrate unexpected talents.

For a project longer than three weeks, look at FAIR PROJECT GRADING.

❋❋❋

This pattern has been used for a group project to produce web pages as part of an introductory computer information systems class, a group programming project, and a group database design project.

Figure 14: « One Grade for All » Pedagogical Pattern

One aspect is worth noting is that, although there are already some authors proposing different strategies for applying patterns in the e-learning area, many existing pedagogical patterns have been conceived for addressing classroom situations (in-person classes, seminars, etc.), where instructor and students have a rich interactive situation. So, there is a vast field to be worked on, serving to adapt existing patterns and to propose new ones applied to the online world. Also, for a non-exhaustive list of class-based pedagogical patterns please refer Pedagogical Patterns Project[7].

2.10.1 Designing Learning Scenarios with Pedagogical Patterns

In the area of pedagogical strategies, the Pedagogical Patterns project has compiled a series of pattern collections applicable to learning strategies. Being the main idea of pedagogical patterns the "harvesting" and documenting of successful experiences in designing teaching

[7] Please refer to the site www.pedagogicalpatterns.org.

strategies, patterns are closer to informal representational structures, like use cases (Booch et al., 1999) or scenario-based design (Carroll, 1995) than to the more formal models. That is, patterns are a more useful approach for teachers who, not being necessarily skilled in design language, want to have their strategies documented in the problem-context-solution format proposed by patterns.

The advantage of such approach, though, is that patterns do not present any constraint imposed by fixed vocabularies and, as such, do not fit into a pre-established design language – on the contrary, they define one. Therefore, they can be used as a first step in the design of learning scenarios, which will possibly encompass other formal notations in later stages of development (in fact, the prototype described in Chapter Five will offer an environment which will facilitate this process). And designers can be assured that if they manage to capture the gist of the pattern, they will have in hand something that was tested before and that works for the context it was made for.

When designing a learning scenario according to the *patterns* principles, instead of thinking in terms of sequences of activities to attain an objective – as it was the case of IMS-LD -, learning designers would rather be guided by the patterns' sections. To start with, they will certainly consider the aspects that might have contributed to the poor results obtained in a previous execution of the scenario (i.e., the *problems*). Then, designers would try to elicit all elements that did have an impact on the learning situation (i.e., the *context*). Once both problem and context are well understood, next step would be to look for proven strategies to solve the problem in question and apply it (i.e., the *solution*). At this time one would want to refer to a pattern repository from the domain of interest.

As already stated, a pattern can also give rise to other representations of the same scenario. For example, if, later, the scenario author decides to represent a given pedagogical pattern scenario in a formal language, he/she would have to express the pattern text using the target language's constructs. In the case of IMS-LD, for example, these constructs would be activities, roles, role-parts, methods, etc. A more detailed description of a translation between pedagogical patterns and IMS-LD can be found in (de Moura Filho & Derycke, 2005).

2.11 Conclusion

This chapter addressed some questions concerning the design of learning scenarios. For that, I discussed a few fundamental principles of design and suggested how these principles could be applied to the domain of education, showing some examples of how this is actually done.

In particular, I was interested in the representations of learning situations that could be used for the purpose of designing learning applications. I showed, for example, that scenarios lend themselves to this purpose, but also that they bring an implicit temporal component that makes them inappropriate as a metaphor for many learning situations in which time is not of paramount importance.

Another problem detected in the IMS-LD approach is the impossibility of adaptation of its vocabulary to real situations. In their everyday practice, practitioners come across unexpected situations that may require adding new concepts in their models, should they represent them as first-class elements. With a fixed standard vocabulary this becomes almost impossible.

Next chapter I propose an alternative approach to designing learning scenarios, which overcomes some of the scenario's shortcomings mentioned here. However, to implement it, a technology is needed, that permits to go automatically from a conceptualization to a running application and that enables changes in the conceptualization to be reflected into the application and possibly, manipulated by domain experts. Such a technology will be discussed in Chapter Four.

Chapter Three: A Multi-model Approach for Representing Pedagogical Scenarios

3.1 Introduction

There is a multitude of teaching and learning scenarios used in real educational situations, and, as we have seen in the Chapter Two, Educational Modeling Languages (EML) are languages designed to capture them. One of the aims of this chapter is then to study the representation of real learning scenarios with one specific EML. Also, by analyzing the consequences of adopting one specific model over another, I investigate the possibilities of using different models to represent a given teaching/learning situation.

I argue in this thesis that we must be wary of the far-reaching impact of choosing a specific conceptual model to represent such scenarios. I suggest in this chapter that, instead of looking for integrative models that aim to represent entire domains, designers would be better off by putting emphasis on the expressiveness of the models and their significance to the actual communities for whom they are designed, even if that means to use different models for specific purposes.

In fact, one of the main contributions of this thesis is to devise an approach that advocates that, instead of having a single EML to represent all possible learning strategies, learning designers should allow multiple models to co-exist, so they can provide complementary insights for learning phenomena -- or simply have the possibility to choose the one that best suits their needs for a given situation. Additionally, in Chapter Five, an implementation is described that realizes this approach.

The present chapter presents this *multi-model* approach for designing learning scenarios, describes its advantages and disadvantages, and argues why it could be a better technique for

building constructivist scenarios -- and post-modern scenarios[8] in general. The rationale here is that, since we do have different paradigms when it comes to interpreting educational phenomena, we should take that in account and allow for multiple models, if we are to have any chance of successfully representing these phenomena.

I base my arguments on some criteria designers use to build their models, like *expressiveness*, *neutrality*, *specificity* and the role of *context*. Also, I address the question of *standardization*, which plays a crucial role in many different domains, but which, if it fails to take account of the plurality of paradigms in such a complex domain as education, risks over-legitimizing a single perspective at the expense of other possible interpretations of educational phenomena.

3.2 The quality of conceptual models

In the previous chapter I have broached the role of models in the design theory and particularly of conceptual models as ontological answers to design problems. In this section I will set about discussing what makes a model to be a good model, giving a special emphasis to educational models, or educational modeling languages (EML).

The quality of a model is a direct consequence of a number of specific decisions taken during the design of the model. To this aim, (Gruber, 1995) proposed a set of design criteria for ontologies – but that are equally applicable to models or other types of conceptualization formalisms - to be used for guiding and evaluating designs. His criteria are:

1. Clarity: "An ontology should effectively communicate the intended meaning of defined terms." That is, definitions should keep the possible number of different interpretations of a single term to a minimum.

2. Coherence: definitions should be logically consistent; that is, there should be no inferences contradicting the axioms of a model.

---

[8] *Post-modernism* here alludes to the movements in different domains of human activity, such as philosophy, art, education, science, etc., that intend to replace or extend the most valued ideals of *modernism*, like rationality, objectivity and progress. While in some of these domains, notably the 'hard' sciences, post-modernist ideas are still very controversial -- though they have many vocal advocates --, in others, especially social sciences -- including education --, the dominance of the *modernism* has receded so that its "mantle of hegemony has in recent decades gradually fallen on the shoulders of the post-positivists" -- or post-modernists. An expression of the post-modernist ideas in education, according to (Wilson, 1997), is the Constructivism.

3. Extendibility: An ontology or model should accept *monotonic* extensions and specializations – i.e. the addition of new premises to antecedents does not invalidate their previously reached conclusions.

4. Minimal encoding bias: The conceptualization of a domain should not be affected by the concrete syntax used to represent it. In other words, the semantics of a model is completely defined in its abstract syntax.

5. Minimal ontological commitment: A model should define a minimum number of concepts necessary for the communication of knowledge within a domain, i.e. it "should make as few claims as possible about the world being modeled".

However, while having the above criteria in mind, I suggest including further criteria, namely the *specificity, expressiveness*, *neutrality* and the *role of context,* to analyze how design decisions can impact on the quality of models. The next sections are devoted to the definition and discussion of the role of these additional criteria to the design of models, putting a special emphasis on educational metamodels, or educational modeling languages (EMLs).

3.3 On the specificity of models

Conceptual models can be more specific or more generic. Generic ones are "wider" in the sense that they are applicable to a larger universe of domains, but they are shallower, usually describing only superficially these domains. Conversely, specific models are tinier scoped, but go "deeper" – or at least should go -, usually describing details with a semantic precision not achievable with generic models. "Intuitively a more specific ontology is indicative of high quality of knowledge" (Supekar, Patel, & Lee, 2002).

3.3.1 A definition for specificity

For the purpose of this thesis, the specificity of a conceptual model is the degree of detail and precision contained in the model. That is, it represents a measure of how ambitious a model is in its aspirations to precisely describe a given domain and reflects how much knowledge and at which level of specialization the modeler intends to represent the domain.

Note that the concept of specificity as defined here is connected to that used in software engineering, which captures the notion of *scope of an abstraction.* Specific models present reduced scope, that is, a more specific abstraction solves a smaller number of problems (Greenfield & Short, 2004).

The correct level of specificity will maximize advantages and minimize disadvantages, or, as (Jackson, 2001) puts it, the value of an abstraction increases with its specificity to some problem domain. Note also that everything said in the preceding paragraph about the specificity of models equally applies to other representational formalisms, like languages, protocols, theories, etc.

3.3.2 About standards and specificity

Standards can be defined as "documented agreements containing technical specifications or other precise criteria to be used consistently as rules, guidelines, or definitions of characteristics, to ensure that materials, products, processes and services are fit for their purpose" (ISO, 2005). Whether achieved through a simple agreement or through enforcement by national or international standard bodies, an standard's distinguishing feature is its uniqueness in the specific domain it applies to. Thus, if modeling means choosing, for a specific purpose, the right concepts, their properties and relations to represent a given domain, then standardization assures that this particular choice can be shared by all communities working on the same domain.

The advantages of the use of standards in the domain of learning scenario design have already been pointed out:

> In the context of e-learning technology, standards are generally developed for use in systems design and implementation for the purposes of ensuring interoperability, portability and reusability. These attributes should apply to both the systems themselves and of the content, data and processes they manage (Friesen, 2004).

In fact, the "ilities" listed above are usually attributed as the benefits reaped from the adoption of standards not only in the "context of e-learning technology", but in any domain where they are used. However, standards also present disadvantages. (Megginson, 2005) enumerates a few of them:

1. FUD: The existence of a real or proposed standard, or even the announcement of a plan to create one, can throw an entire industry into Fear, Uncertainty, and Doubt.
2. Monoculture: Standardization discourages diversity, which is the best defense against design flaws and security vulnerabilities.
3. Square pegs: When a standard is successful, the pressure to apply it in inappropriate places, where its disadvantages far outweigh its benefits, can be enormous.
4. Abstraction: Although abstraction is sometimes helpful, ignoring lower-level issues can sometimes lead to large inefficiencies or even rule out simpler and better solutions to problems.

The second point above is the one I would like to put the focus on now. As explained in the previous chapter, along the years, communities put a lot of effort into designing and refining their knowledge base, documenting it in the form of models, languages and other representational artifacts. The standardization process of such artifacts is a natural consequence, once an optimal level of specificity is found and if there is enough interest from the part of a critical mass of users wishing to reap the benefits of its use *en masse* (sharing, collaborating, etc.)

But incipient domains tend to develop immature and unspecific (i.e. high-level) conceptualizations that may not conveniently address real world problems, which are not fully understood. Succumbing to the temptation to release standards prematurely means combining unspecific models with the mono-cultural aspect introduced by standardization processes, what will serve only to homogenize varnish-deep knowledge. This can be further aggravated by the "congealing" power of standards, which will cause any "design flaw" to endure for as long as the standard is in force. Even at the pre-standard stages (e.g. specifications) this can do harm, since there are always early adopters, who, willing to take advantage of bleeding edge technology, have no time to appraise the convenience of these models.

TCP/IP: a case study

When explaining the advantages of standardization, it is commonplace to recite some examples of popular *de facto* or *de jure* standards, like DVD, MPEG or the Internet Protocol.

I would like to elaborate here a little bit more on the example of the IP as a universal standard in order to introduce some questions concerning the specificity of models[9].

In fact, the Internet would be impossible, as we know today, without the convergence toward a single inter-network protocol, in that case, the Internet Protocol (IP). Incidentally, other protocols could also have been chosen instead of IP, as long it fulfilled the requirements specified by the OSI's Network layer, notably that of end-to-end package delivery -- which requires capabilities like translating logical addresses and names into physical addresses, calculating routes, managing data congestion, etc. The point here is that, once a given network protocol is adopted, it must be unique.

IP is therefore a typical example of the advantages of having a single standard[10] at the planetary level and the price to be paid for not being compatible with it is to stay out of the Internet. However, the very Internet protocol stack provides us with other examples of the need for allowing choices when a single standard no longer satisfies conflicting requirements from different "users" – in this case, end user applications.

Right above the inter-network layer, for example, roughly corresponding to the OSI Transport layer, the TCP/IP stack offers two separate transport mechanisms[11] for user applications. On the one hand, for those applications requiring a reliable transport mechanism, there is the TCP protocol with all needed techniques (like, three-way handshaking, positive acknowledgement, retransmission upon timeout, sliding windows, etc.) to guarantee that the transmitted data will arrive at the other end without error.

On the other hand, some applications, notably audio and video streams, which don't need ordering or reliability guarantees –at least not as much as they do need throughput and delay guarantees - usually opt for the UDP protocol. That is, these applications prefer a protocol

[9] I could have illustrated just as well this idea with any other widespread protocol, but, given my background of networking assistant teacher, I could not resist the choice for the TCP/IP.

[10] For the purpose of this analogy, I decided to gloss over details like the existence of two versions of the IP protocol, IPv4 and IPv6, as well as other protocols that are technically part of the OSI layer 3, but that have different purposes, like ICMP, IGMP, ARP, RARP and the different routing protocols.

[11] Once again, for the sake of simplicity, I let aside other protocols residing in the transport layer of the TCP/IP stack (like RTP, DCCP, SCTP, IL, RUDP, etc), on the basis that they present overlapping functionality with either TCP or UDP.

that is unreliable – i.e. offer no guarantee of the delivery of data entrusted to it – but that is lighter and faster (exactly for not implementing TCP's connection-oriented flow control and error control mechanisms). Thus, the TCP/IP protocol suite, in its transport layer, does offer alternatives for "users" reflecting different requirements – more than that, reflecting on different paradigms.

A more radical example of allowing for multiple choices is provided by the TCP/IP stack once we move one layer up, now entering the OSI Application Layer. In this layer we are going to have a huge and ever growing number of protocols – like FTP, HTTP, SMTP, etc. – each offering tailored services to very specific purposes. That is, in order to satisfy very precise –and many times incompatible – requirements from different applications, several specific protocols were deemed necessary.

Addressing the needs of the application layer with one single non-specific protocol would mean to have a protocol that met the different needs from all user applications on top of it. Such approach would yield either an extremely complex protocol, unwieldy enough to address any single application needs, or a protocol that, even if kept at manageable proportions, would fall short of addressing the particularities of the applications. That is why this approach could never be seriously considered for the application layer.

The TCP/IP protocol illustrates, thus, the whole range of possibilities for different specificities:

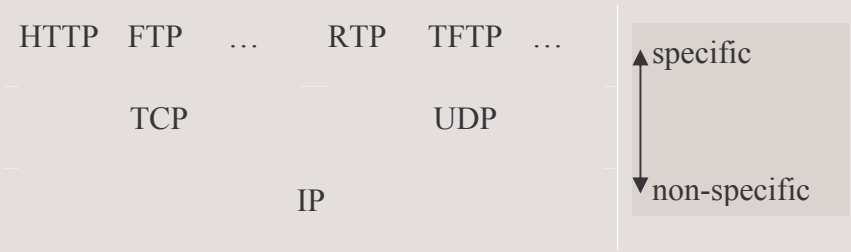


Figure 15: the TCP/IP protocol stack

The bottom-line of the TCP/IP example is that -- and networking multi-layered protocols helps us see this more clearly -- the closer to the human user, the more *specific* and *pluralistic* are the alternatives that must be offered, so as to address the multiple and distinct needs so

typical of humans. This is a lesson that the mature networking community, which developed TCP/IP, leaves to other domains wishing to standardize their "protocols" – i.e. their models, their languages and ultimately their knowledge.

Another example that could be cited here is the one provided by the domain of ontologies, which offers low-specificity ontologies (i.e. upper level ontologies, which cut across all possible fields) as well as very specific ones, applicable to only tiny scoped domains. Upper level ontology, with coarse grained constructs like *concept*, *state*, *event*, *process*, *action*, *component*, etc. cannot properly represent phenomena of individual domains with their specific needs.

To provide more substance to this discussion, I will need to introduce another feature of representational artifacts in general, which is the *expressiveness*. But before this, still in this section, I will discuss the problem of specificity applied to the domain of the educational modeling languages (EMLs).

3.3.3 Specificity and EMLs

A question that naturally ensues from the above discussion about specificity is if a given domain can be addressed with one single – and unspecific - model or if it is complex enough to entail subdivisions, each one with its own highly specific model. For this thesis, I am interested in the domain of learning scenarios, so, the question then becomes, "is it possible to model all possible learning scenarios with one model only"? If not, how many models or languages should learning scenarios admit and what is the degree of specificity of each? That is, the issue of what *is* the correct specificity of the learning scenario domain's model may lead directly to another issue, concerning the multiplicity of models.

First I will consider the approach of having an unspecific, integrative EML. That is, like in the case of the upper-level ontology, described above, we have one single model with a few concepts to represent all possible situations that may appear in the domain.

I consider that one of the biggest challenges for this single model would be to traverse the chasm separating scenarios drawing on different paradigms. Kuhn, to whom we owe the contemporary meaning of the word paradigm ("an entire constellation of beliefs, values and techniques, and so on, shared by the members of a given community" (Kuhn, 1962)) argued

that the proponents of different scientific paradigms cannot make full contact with each other's point of view as they are, so to say, living in different worlds. This claim became known as the “thesis of incommensurability of paradigms” (Kuhn, ibidem).

I suggest that it is very difficult – if possible at all – to create a single metamodel that crosses the boundaries of paradigms and correctly represents phenomena as expected by their corresponding worldviews. Put differently, two paradigms that are mutually exclusive (the paradigms underlying the worldview of constructivists and behaviorists, for example) will not be open to simultaneous use if we have only a single model available. In fact, the “incommensurability of paradigms” would undermine any effort of common representation, since now it could happen that the model chosen to represent a given phenomenon has absolutely no relevance for the problem in question, when viewed through the conceptual framework of a “rival” paradigm. Or, to use Kuhn’s words, “paradigmatic differences cannot be reconciled.”

Thus, where there is the need to create representations for phenomena to which there are multiple interpretations drawing on incommensurable paradigms, we should consider creating multiple models instead. Of course good sense should always guide us through the interpretation of this affirmation – or of any other. If the modeler’s particular intent is to contrast two paradigms, he or she would have the need of representing concepts from both paradigms. As an example, in a would-be model crafted to compare the views of theologists and cosmologists about the origin of the universe, these two paradigms would be present – for example, making the concept of “God” a *boolean* type, so as to accommodate both arguments.

The possibility of having complex enough domains that encompass different theories drawing upon antithetic paradigms, takes us to the other side of the *specificity* spectrum, which is the one of allowing multiple highly specific models.

So I would like to consider now the possibility of having multiple metamodels to address the different aspects of the learning scenario domain. This would reflect the idea of different users having different needs, interests, perspectives, etc. – what is not so uncalled for in sufficiently complex domains. It seems clear to me that learning scenarios belong to this latter category, considering the rich diversity of strategies in the field of teaching/learning and the multiplicity of paradigms that they stem from.

Therefore, when creating representations for the domain of learning scenarios, I argue that, instead of using one single integrative metamodel for the whole lot of pedagogical strategies, we should use one ad hoc metamodel for each specific strategy. For example, we could create a metamodel for the Problem-based learning (Boud & Feletti, 1991) strategies (incidentally, this has already been done by (T. Nodenot & Laforcade, 2006)), another for, say, the jigsaw strategy, still another for the Peer Feedback pedagogical pattern (as sketched in the previous chapter), etc.

Since the aim of this section is just to introduce the importance of specificity for the creation of models, I wrap up here the discussion of specificity and multiplicity of metamodels. However I will return to this subject later on, once three other important criteria to be considered when building models, *expressiveness, neutrality* and the *role of context,* have been introduced.

3.4 On the expressiveness of models

While specificity translates the idea of the *range* of a representational artifact, expressiveness is related to its *adequateness* to the domain it is supposed to represent, better expressed on how comfortably can experts from the domain express their ideas. That is, if specificity is characteristic of the *structure* of the representation artifact, expressiveness will be manifest in the process-centric design, by allowing experts to express exactly what they mean. Furthermore, specific representations, as a rule, tend to be more expressive about the small universe they deal with.

3.4.1 Initial considerations

The Merriam-Webster's Online Dictionary defines *universe of discourse* as "an inclusive class of entities that is tacitly implied or explicitly delineated as the subject of a statement, discourse, or theory." For the purpose of this work, thus, a model is said to be expressive for a given domain if it presents constructs semantically equivalent to each of the entities (concepts, grammar rules, axioms, etc.) found in the domain's universe of discourse. That is, its types, syntax and operations map directly to the respective elements from the domain.

Trying to model a domain with a metamodel that is not expressive enough is exactly what (Megginson, 2005) meant by "square pegs [in round holes]". That is, the more specific domain concepts must be "cast" – if they can be conserved at all - into the metamodel's constructs, sometimes with no exact semantic equivalence. This procedure will eventually lead to "flattened out" representations in which the idiosyncrasies of the domain are lost. In other words, different situations, which in real life are completely distinct, will be homogenized under the same concepts, thus looking like they belonged to the same category.

(Fields, 2001) addresses the same subject in his thesis:

> When the language is defined, many of the choices about what to leave in and what to leave out, and therefore what can be said, are made ahead of time by the constraints of syntax and semantics. One of the effects is to obscure the choices that are being made, and make the kinds of description that result appear 'natural' and 'neutral'. Or to put it another way: When all you have is a hammer, every problem looks like a nail.

Thus, the main advantage of expressiveness of models is letting different things look different. If *"research is a strenuous and devoted attempt to force nature into conceptual boxes"* (Kuhn, 1962), the same can be said about a model. In that case expressiveness is the guarantee that the boxes are large and varied enough to hold all relevant ideas.

Of course there are always trade-offs in trying to create highly expressive models for complex domains. For example, adding concepts in an effort to capture all aspects of complex phenomena could make the model unwieldy for tackling any particular design question. Therefore, for any reasonably rich domain, I consider that a good way of increasing expressiveness is breaking it down into more specialized sub-domains and tackling each separately – which also means increasing specificity, as discussed in the previous section.

3.4.2 Expressiveness and EMLs

For the specific domain of pedagogical strategies, (van Es & Koper, 2006) provide a definition for expressiveness:

> Pedagogical expressiveness is defined as the ability of a modeling language to describe all types of teaching-learning situations (pedagogical flexibility) including the needed flexibility to adapt the UOL to predefined criteria or situational circumstances (personalization). The modeling language must be

> able to describe all learning objects that occur and their relation with the teaching-learning process (completeness).

This definition rephrases, for the field of pedagogical scenarios, the idea of having a way to represent the entire domain of discourse – or "all types of teaching-learning situations". The problem, however, based on the arguments from the previous section, is if this can really be accomplished using one single model/language, given the high complexity of the domain of pedagogical strategies.

In my opinion, it is highly unlikely that the generated instances of such a language will faithfully represent each of the strategy's intentions. It should be clear by now that being able to accommodate any type of scenario with its constructs is not a good enough indicator of the expressiveness of a language. For example, we could just as well have the same scenarios represented using the UML metamodel (OMG, 1997) or any upper level ontology and that wouldn't make of the UML or of the ontology expressive learning scenario modeling languages.

To summarize, we cannot draw any conclusion from the fact that we can pinpoint terms from a scenario description and cast them to certain language constructs. It can be that in the process, for the lack of significant concepts of the target model, we overlook most of the scenario idiosyncrasies, thus missing much of its original intent.

Furthermore, using a single language to represent complex domains introduces another more subtle problem, which reflects the fact that such a "multi-dimensional" language, in addition to - purportedly – accommodate different paradigms, belongs itself to a specific paradigm, raising the question of *neutrality*. (Allert, 2004) addressed the issue -- which she called *modeling diversity* -- explicitly for the domain of education:

> On the one hand a modeling approach has to be able to describe divers concepts of learning, which refer to different epistemological and ontological positions, on the other hand the modeling approach itself reflects a specific epistemological and ontological position. How can an approach which reflects a certain position describe learning concepts which reflect another?

I will give more arguments to answer this question in the next section, which makes an appeal to the theme of the neutrality of models.

3.5 On the neutrality of models

3.5.1 Models as biased artifacts: what lies beneath…

Perhaps influenced by the positivist "received view", which advocates a single and objective reality, people are naturally inclined to accept the idea of having one single model as a mandate entity representing this unique reality. However, doing that means overlooking the fact that models are not "transparent windows on an independent reality", but rather human artifacts that serve specific purposes, needs, priorities and interests.

To this aspect, it is never unwarranted to remind that the constituting elements of a model are not there by chance, but have been *put* there. Not being able to rephrase this idea as eloquently as (Robinson, 1997), I prefer to quote him verbatim:

> In one of the first substantial attacks on mental images as 'little pictures in the head', Sartre used the well-known image of the Parthenon. His challenge, for those who have visited or otherwise know the Parthenon, is to conjure up a mental image and then count the number of pillars.
>
> Of course, this cannot be done unless one knows the number in the first place. Images are not like photographs. One only puts into them what one already knows. Images are produced in active consciousness; they are not things. This chapter suggests that the same holds true of models, that is, that the main utility lies not in what is there, but in the activity of putting there.

This idea of models – just like any other human artifact - being inextricably biased is not new and is a recurrent one in scientific works. (Feyerabend, 1975), for example, advocated the theory-ladeness of scientific methods, by affirming that they are neither commensurable nor objective in the evidence they provide. Also, from a different prism, (Fields, 2001) discussed the impossibility of creating neutral models:

> The processes of crafting models and representations therefore must be considered as human activities that are part of a complex web of interconnected forces and interests. [I]n fact models embody a particular view on the modeled or represented phenomenon, and reflect a series of choices and judgments that have been made about it.

Not surprisingly, these assertions corroborate the idea of models as biased artifacts that refer to only some aspects of the phenomenon in question, and discredits the positivist naïve view that they have been created by a "disinterested scientist", who is as a mere "informer of

decision makers, policy makers, and change agents" (Guba & Lincoln, 1994). Rather, they posit relevance and engagement over disinterested academic observation, assuming that "designers act in purposeful, value-based ways with ethical knowledge, in social relationships and contexts that have consequences in and for action" (Campbell, Schwier, & Kenny, 2005) – even if it may sometimes degenerate into indefensible, agenda-driven research, which attempts to massage favourably or coax the results of experiments for purposes which are often far from acceptable.

3.5.2 Neutrality and Educational Modeling Languages

While recognizing that learning theories are not pedagogically neutral, communities that develop standards for the learning industry share a common assumption that reference models, like IMS-LD, and metadata specifications, like LOM, are neutral, *"as they are integrative meta-theories"* (Allert, 2004). Two manifest expressions of this assumption are transcribed below:

> This standard will specify the syntax and semantics of Learning Object Metadata, defined as the attributes required to fully/adequately describe a Learning Object (LOM, 2002).

> The reason for developing a metamodel was to have a model that was neutral with respect to different approaches of learning and instruction. Neutrality in this context means that specific pedagogical models, like problem-based learning models or collaborative learning models, should be able to be expressed using the metamodel with the same ease (van Es & Koper, 2006).

I acknowledge at once my skepticism about this view for the reasons described in the last sub-section, namely, no human artifact can possibly be neutral. For example, we are reminded by (Allert, *ibidem*) that, with LOM, "it is almost impossible to annotate relevant structural elements of situated learning and innovative learning with attributes and categories defined in LOM." For her, LOM is more apt to describe resources using an acquisition metaphor and she then grounds her assertion on the example of LOM specifying such attributes as *Typical Learning Time and Semantic Density* (within its "Educational" category):

> It assumes that the semantic density is determined by the characteristics of the resource: enclosed in the learning object and transferred to the user but not constructed by the user.

In that case a learning object containing a poem by Shakespeare would be tagged with a fixed "semantic density", unbeknownst to whether it will be used in a course on literature or on literacy.

Allert takes then a bolder stance and concludes:

> [T]here is no chance [of] being neutral, as referring to an epistemological and ontological position is unavoidable. Defining the structure of metadata and specifying a conceptual data schema inevitably reflects a specific concept of knowledge and meaning.

Considering models as vehicles of interests, it is important to have in mind the context -- and hence interests -- in which those specifications appeared. That is, once we recognize the value-ladenness of models, it is important to elicit the values behind them, because, as (Wilson, 1997) puts it, "valuing one perspective means that other perspectives will be given less value". And maybe a more considered analysis would have taken us to choose one of the neglected perspectives. To this subject, Wilson (ibidem) suggests a few questions to be asked in order to uncover the values embedded in instructional designs:

- Who makes the rules about what constitutes a legitimate learning goal?
- What learning goals are not being analyzed?
- Whose interests does the project serve?
- What is the hidden agenda?

In the next paragraphs I will lead an analysis in an effort to answer these questions for the example of the IMS-LD, though the reader will soon realize that this analysis -- and the whole thesis, for that matter -- has its own values, which do not always coincide with those advocated by the IMS-LD supporters.

I am going to develop an analysis of IMS-LD's needs and interests from two perspectives: a business perspective and an educational perspective. To understand the business perspective we can take a look at its main sponsors: IMS Global Learning Consortium and Open University of the Nederlands (OUNL). I let them speak for themselves:

> IMS Global Learning Consortium is a global, non-profit, member organization that provides leadership in shaping and growing the learning industry (IMS, 2006)

> Open Universiteit [is] an institution that operates successfully in the field of lifelong learning and is a much wanted provider of market-oriented and commercial education. It is a frontrunner in open higher distance education and a leader in educational innovation, also on an international scope (OUNL, 2007).

Assuming that IMS-LD reflects the *business* model adopted by their sponsoring institutions, we can apprehend from the paragraphs above and from the website of both institutions that this metamodel will most likely meet the needs and interests of:

- Training programs, as opposed to a more complete education.
- Big institutions, with thousands of students.
- Quality-control regulated courses, emphasizing repeatability, measurability, etc. so as to assure that every student will have the same quality of training.
- Standards-enforced solutions (as opposed to ad-hoc ones) so as to take advantage of the reuse of resources.
- Highly valued commitment to technological and economical aspects, paving the way to "cost effective education" and "(commercial) distance and e-learning training programs".

While recognizing that the "learning industry" approach is useful in some situations and that epitomizes the business model of many institutions, especially large universities, we cannot expect it to be a good match for many others, especially smaller and public schools. In particularly it does not seem fit for institutions that do not share the philosophy of wholesale and commoditized education – preferring ad hoc solutions to more contingent needs - or those which simply cannot afford it (because the "learning industry" is only cost-effective for large-scale "productions"). From this perspective, the strict adherence to standards, highly recommended by standards committees, is clearly market-driven. This is also true of the education domain:

> The market also demands "standards" in non-technical areas, such as "education standards" (ISO, 2002)

As for the *educational* perspective, it is tantamount for learning designers and practitioners to elicit the values and interests of IMS-LD's metamodel, so they can consciously verify whether these values match their owns and can decide whether or not to adopt it. For example, when designing learning scenarios, as we have seen in the last chapter, IMS-LD

designers are compelled to think in terms of roles, learning objects, etc. and then to arrange everything in a *theatrical play*. Although it uses the metaphor of a play to better convey the image of a Unit of Learning to non-computer-savvy users, IMS-LD metamodel has been predominantly couched in – and can be better understood in the light of - the paradigm of a workflow[12]:

> The formal language must be able to fully describe a unit of learning, including all typed learning objects, the relationship between the objects and the activities and workflows of all students and staff members using the learning objects. (Van Es & Koper, 2006)

This metaphor is not anodyne and much less neutral, since it may rule out any possibility of expressively representing scenarios that cannot be faithfully described through temporal sequences of learning tasks and conditional control transfer logic. As explained in the previous chapter, there are many timeless concepts that may be important for a given learning experience but that cannot be plotted in a time frame – or simply does not make sense to do so.

The other metaphor behind IMS-LD is that of a scenario. IMS-LD offers a generic framework serving to create automated scenarios and, as an implementation of the scenario metaphor, inherits its benefits and limitations, as clarified in the preceding chapter.

Also, since IMS-LD declines to represent terms stemming specifically from the education domain, it could be used in any situation where we have roles assigned to people, performing activities using resources (e.g. in the domain of Business). There is nothing in IMS-LD preventing its use with other scenario-like situations, beyond educational scenarios. This lack of specificity is acknowledged by the IMS-LD authors:

> "The need for a vocabulary of pedagogical approaches therefore lies outside the requirements for the runtime implementation of the designs, and the task then is to elaborate this need more clearly, identifying the actors and their use-cases. […] For these kinds of purpose, a taxonomy of learning approaches or pedagogies would be needed for use in the IMS/IEEE Meta-Data <classification> field."

---

[12] **Workflow** at its simplest is the movement of documents and/or tasks through a work process. More specifically, workflow is the operational aspect of a work procedure: how tasks are structured, who performs them, what their relative order is, how they are synchronized, how information flows to support the tasks and how tasks are being tracked (Wikipedia on Workflow).

That is, the only way for a UoL to communicate any semantics of a given learning theory or strategy is by tagging it with keywords, which could then be used for search, indexing and storing purposes, for example. However this approach does not help designers to represent the fundamental concepts required by each specific scenario, what may negatively impact on the representation of a learning experience's, falling short of conveying its author's intention.

3.6 Models and context

After having analyzed models and languages in the light of their intrinsic specificity, expressiveness and neutrality, there is one more criterion I would like to add to the discussion: the role of the context. With this fourth element we will be able to more appropriately discuss the adequacy of these representational artifacts for describing real world situations, problems, systems, etc. as well as the question of how they can better represent learning and teaching phenomena.

3.6.1 Learning scenarios and the role of context

We can elicit the pedagogical overtones of different models and languages describing learning scenarios (or EMLs, for short) by analyzing the way how they deal with – or fail to do so – the *context.* In fact, an important distinguishing feature that sets apart constructivist and positivist scenarios is the way in which they consider the role of the context.

I consider in this thesis the definition for context as found in (Figueiredo & Afonso, 2005). By context, they mean "anything that can have an impact on the learning situation" and this is the meaning I will adopt in this work. Although I am aware of the extensive work made in many different areas to formalize the notion of *context,* the above definition, though not rigorously formal, is adequate enough for the intent of this section. For a more detailed view of a *context*, though, the reader may wish to refer to (Bradley & Dunlop, 2005), where a compilation of definitions for the term, coming from areas as different as philosophy, linguistics and computer science - among others - has been provided.

According to the positivist paradigm, context is "external and clearly independent of the learner and the activities in which the learner is engaged", presenting such a stable and

predictable behavior, that it can be "characterized in advance". They also add an example of how positivists think of context:

> [W]hen developing content for a given course, we take context into account beforehand in the elaboration of our materials, and we then forget about it, trusting that its behavior will always be as expected. (Figueiredo & Afonso, ibidem)

That is, positivist visions of learning tend to ignore or simplify context, "*disregarding its absolutely unavoidable social dimension."*

As for the constructivist paradigm, still according to Figueiredo and Afonso (ibidem), a context follows the phenomenological hypothesis, and therefore can only be *"perceived through its interactions with the learner",* while changing permanently.

3.6.2 A schema for analyzing context in educational modeling languages

When working with EMLs, it helps to consider that the context of a learning scenario is in fact composed of (at least) three different parts, each contributing - to a greater or lesser extent - to the overall context:

- one part is defined during the conception of the chosen EML (which I call here the *language context*);
- another part is defined by the designer during the conception of the scenario itself (the *designer context*)
- and the last part, established during the "execution" of the scenario, which is eminently learner-dependent (the *learner context*)

The *language context* is the 'hard-coded' context, which was defined by the EML authors (or, using the terminology from Chapter Two, the upstream learning designers) long before the (downstream) learning designers have even thought about expressing their particular scenarios using the modeling language in question. It is the context imposed by the fixed vocabulary of the chosen EML.

The *designer context* is the scenario as perceived by the learning designer, at design time. While the EML defines the *metamodel*, here we have the *model* that is created by grouping, in

any particular arrangement, the constructs of the chosen EML. For example, creating a learning scenario with IMS-LD, in practical terms, means that the designer has to characterize in advance, not only the activities that will be executed by learners and the "surrounding" environment (e.g. learning objects and services), but also the strict temporal sequence in which each of these elements will come into play.

Finally, there is the *learner context*, which encompasses everything else that was not specified by the other two contexts and that can have an impact on the learning situation, as perceived by the learner, including activities *actually executed* by the learners (as opposed to those anticipated by the learning designer), interactions among participants, socio-economic and physical conditions, etc.

3.6.3 Context and the IMS-LD specification

In the light of the schema above, I can now go about to analyze the way context is represented in an IMS-LD Unit of Learning (UoL). As stated in (IMS-LD, 2003), a UoL is assumed to be completely and fully described, according to the requirement about completeness of the IMS-LD information model[13]. I sustain that IMS-LD, by putting a strong emphasis to the "anticipated context" (i.e., language context plus designer context) seems manifestly more adapted to pedagogical strategies drawing upon the positivist paradigm. That is, the idea of having de-contextualized learning scenarios, which are specified once and then reused in different situations, seems more adequate to an instructivist scenario (knowledge transfer metaphor) than to a socio-constructivist one (shared experience metaphor).

Of course this is not specific to IMS-LD, but to any situation where design (of both model and metamodel) and execution are distant enough in any possible dimension – time, space, socio-economic, cultural, etc. Nevertheless, it has already been suggested that the missing ability to describe the relation between the learning design and its context is in fact *"a blind spot of activity-centered models"* (Scheunpflug, 2001 - *apud* Allert, 2004).

Careful learning designers wishing to represent constructivist scenarios might want to choose more targeted EMLs and to keep design-time constraints to a minimum, so as to leave a great

deal of freedom to the creative initiatives that will naturally appear during "execution time" - eventually giving shape to the overall context. That is, the designer should surrender control over the scenario, and in so doing, he or she *"participates more meaningfully in the total design of the experience"* (Wilson, 1997).

3.7 A multi-model approach for designing learning scenarios

The previous sections of this chapter have discussed concepts like expressiveness, specificity, neutrality and the role of context in representations. In this section all these concepts will concur to help us derive some conjectures about the design of learning scenarios and further guide us into looking for a solution compliant with these conjectures.

3.7.1 A definition for multi-modeling

From what was exposed in previous sections about specificity, expressiveness, neutrality and the role of context in the different representational artifacts, I formulate the following conjectures:

1. Some domains are too rich to admit expressive all-encompassing models.
2. Models belong to paradigms and cannot traverse them.
3. There are no neutral models.
4. Multiple perspectives can afford complementary insights of phenomena.
5. Models should be as expressive as possible of the context in which modeled phenomena happen.

Considering all these conjectures, I conclude that, to better represent complex domains, we should abandon overarching integrative models in favor of multiple targeted ones. In other words, we should consider building multiple narrowly focused metamodels where integrative standards fall short of expressively represent the relevant concepts of a domain. This is what I call a *multi-model approach* or, when specifically applied to the learning scenario domain, a *multi educational modeling language approach*, or simply, **Multi-EML**.

---

[13] "The specification must be able to fully describe the teaching-learning process in a unit of learning, including references to the digital and non-digital learning objects and services needed during the process." (IMS-LD Information Model).

To be as expressive as possible of the situations that each model is supposed to capture, the creation of each specific model would be postponed to a moment when much of the actual context is known. This means that models should be crafted by learning designers or even domain experts at close range to the modeled learning phenomena.

The multi-model approach would address each of the conjectures from above as follows:

1. If adding concepts to models in an effort to capture all aspects of complex phenomena can make it awkward for tackling any particular requirement, the multi-model approach solves the problem by enabling to break down complex domains into as many parts as necessary to attain an optimal level of specificity and expressiveness. One way of how this can be achieved is by enabling "downstream designers" to create concepts and other constructs of a model with the relevant semantics and at the level of specificity that suits their needs – something that an implementation of the multi-modeling approach should make possible.

2. The multi-model approach assumes that there are multiple paradigms and allows to choose the best adapted model(s) for a given paradigm. In many cases, however, depending on the domain, there is no common understanding about paradigms and their boundaries, or even, as usually is the case, there are orthogonal paradigms, e.g. *subjective* versus *objective* or *regulation* versus *radical change* (Burrell & Morgan, 1979). The multi-model approach, by permitting multiple models and variable specificity, can easily accommodate any variation of conceptualization.

3. With the multi-model approach, the neutrality of models is a question that has no longer importance, since all possible paradigms, each with their values, purposes, ideologies, etc., can be equally represented. That is, it assumes that neutrality is an impossible aim and offers a way for designers to position themselves explicitly.

4. The multi-model approach admits a diverse set of narrowly focused models, each referencing a particular aspect of a whole. For example, for the case of learning design, the same teaching/learning situation would certainly be interpreted differently by experts with different experiences, background, influences, etc. The multi-model

approach would let them express their needs befittingly. That would lead to complementary interpretations with very specific purposes of the same learning phenomenon.

5. The multi-model approach, by allowing models to be constructed close to the "factory floor", possibly by a participant of the learning experiment (e.g. the instructor), can take in account much more of the overall context, if compared to conventional single EML approaches.

3.7.2 Issues raised by the multi-model approach

The multi-model approach raises technical issues, though. Adopting a multi-model approach means that practitioners now have the power to create, improve and maintain their own ad hoc models. Even if powerful tools can be provided to users that leverage what Fowler calls *lay programming* (M. Fowler, 2005) – and this is one of the subjects that will be treated in the next chapter – the multi-model approach also means to stray away from the safe haven of the standard metamodels with their "out of the box" applications to enter a process where practitioners will have to take on much of the tasks once attributed to professional designers and programmers in the standardized model. Some of these tasks are listed below:

- To create multiple models: the multi-model approach presupposes that practitioners will join the design effort. So the question comes down to: "are users capable of creating their own ad hoc models?"
- To reconcile multiple models, i.e. to maintain consistency between different models. This raises the question: "how can order emerge from the apparent chaos of having alternative, competing views for the same system?"
- To execute a model just created. Crafting expressive models is a necessary but not sufficient condition. In addition to their descriptive/prescriptive nature, models must be leveraged by applications (editors, players, databases, etc.), where their semantics will be presented to the final user. Once again, the characteristic of the multi-model approach may signify that a part of the development effort will be transferred to domain experts.

These and other "use cases" will be further detailed in Chapter Five, when I will describe the MDEduc, a prototype implemented to realize the Multi-EML approach.

It is envisaged an extra resistance for the widespread adoption of the multi-model approach from the supporters of standard metamodels. Nevertheless, it is important that professionals be aware of the gains and losses, for their specific case, in adopting standardized all encompassing EMLs.

It is important also to note that the multi-model approach does not rule out standardization processes, but rather proposes a bottom-up approach for it, i.e., with the standardization cycle beginning "at the factory floor", with practitioners trying with new designs, discovering where the real problems are, and then improving them, in a piecemeal growth. Eventually, there will be a natural convergence around user-decided specificity and expressiveness. And that would be the right moment for a standardization process to begin, if there is enough interest from the part of a critical mass of users. This approach would, as an added bonus, address the problem of committees proposing "standards for things that do not even exist, in the hope that implementation will follow and that the problems people discover will happen to be the same ones they anticipated while writing their specification" (Megginson, 2005).

3.7.3 ISD, EMLs and the Multiple EML approach (Multi-EML):

Having introduced the multi-model approach, we are now in position to analyze the differences between the three distinct approaches to learning design discussed in this thesis: ISD and EML, discussed in Chapter Two, and multi-modeling. For this, I am going to avail myself of taxonomies that, though primarily used in philosophy and social sciences, can be very useful to help us understand the evolution in the different approaches. The terms are *etic* vs. *emic* and *nomothetic* vs. *idiographic* and are defined, below (Merriam-Webster online):

- *emic:* of, relating to, or involving analysis of cultural phenomena from the perspective of one who participates in the culture being studied;
- *etic*: of, relating to, or involving analysis of cultural phenomena from the perspective of one who does not participate in the culture being studied;
- *nomothetic*: relating to, involving, or dealing with abstract, general, or universal statements or laws;

- *idiographic*: relating to or dealing with something concrete, individual, or unique;

A more detailed explanation of these terms can be found in the footnote[14].

As seen in Chapter Two, in the *instructional systems design,* or **ISD**, the idea is to develop learning materials and systems according to some educational theories. It follows roughly the lifecycle beginning by an instructional expert devising an instructional theory, followed by an instructional designer conceiving an instructional *design* theory, which incorporates principles of software engineering; then an application is developed respecting the theory and, finally, it is submitted to the final users' scrutiny[15].

In order to adapt the application to their own needs, users can usually change values of some parameters accessible through the UI. The whole cycle – analysis, design, development and test - for bigger applications, can take a few months and even more. When in actual use, if there is any kind of problem that cannot be solved through fine-tuning parameters – for example, when the underlying instructional theory that guided its conception is flawed - the application must pass the whole cycle again. Thus, the ISD approach is a fundamentally *nomothetic* process, where an application is the direct result from a general and "immutable" theory – i.e. it is "hard-coded" in the application and cannot be changed, at least for a particular implementation – and *etic*, considering that experts/developers and users belong to different communities.

---

[14] **Emic** and **etic** are terms first introduced by linguist Kenneth Pike and are used in the social sciences and the behavioral sciences to refer to two different kinds of data concerning human behavior. An "emic" account of behavior is a description of behavior in terms meaningful (consciously or unconsciously) to the actor or actress. An "etic" account is a description of a behavior in terms familiar to the observer. Scientists interested in the local construction of meaning, and local rules for behavior, will rely on emic accounts; scientists interested in facilitating comparative research and making universal claims will rely on etic accounts.

Nomothetic and Idiographic: Coined by philosopher Wilhelm Windelband, these terms describe two distinct approaches to knowledge: **Nomothetic** is based on what Kant described as a tendency to *generalize*, and is expressed in the natural sciences. It describes the effort to derive laws that explain objective phenomena. **Idiographic** is based on what Kant described as a tendency to *specify*, and is expressed in the humanities. It describes the effort to understand the meaning of contingent, accidental, and often subjective phenomena. (Wikipedia, 2006)

[15] As we have seen in Chapter Two, most instructional software is developed according to the ADDIE model – an ID model that stipulates that instructional designers should follow the processes of Analysis, Design, Development, Implementation and Evaluation when developing their tools -- or one of its several spin-offs.

In the **EML**s world, the discourse's focus is shifted from ID processes to a metamodel[16]. Learning designers are no longer obliged to pass through long cycles in order to see their products in action anymore, since they can now create "new applications" just by rearranging the elements of the metamodel in a new model.

This process-centric design will possibly have effects on the different aspects of the final application: new interfaces, new features, etc. can be created that will take only a fraction of the time typically used to develop applications using ID "models". However, and this should be clear by now, they cannot change the vocabulary itself - as it is imposed by a fixed metamodel -, what means that each "new application" cannot significantly differ, both syntactically and semantically, from others derived from the same metamodel.

The life cycle now is: domain experts, "upstream" learning designers and programmers design both the metamodel and a corresponding runtime environment; the "downstream" designer - possibly an expert user - designs the scenario model and deploys it to the runtime platform for the final user to execute. In case any problem appears that cannot be solved on the spot by the user through any specific parameterization, the whole scenario can be redesigned.

While being an inherently etic process -- i.e. the metamodel, hence the vocabulary, was conceived by someone outside the circle where it is actually being used --, the freedom to rearrange the metamodel's constructs allow expert users to add some subjective considerations, by addressing more contingent and accidental needs, as long as expressible with the metamodel, what can makes it a more idiographic approach, if compared with ISD.

Finally, with the Multi-EML approach, I believe we can take one step further, by allowing to change the metamodel - and consequently to modify the very conceptualization of the problem. Here we do not need to have all-encompassing, mature, or even proven theories - though nothing prevents from doing it. In practice, any quick draft conceptualization reflecting very circumstantial needs - maybe realized minutes before actual use - can give rise to an application. This practice reminds the phenomenological models, in that there is not

---

[16] As explained in Chapter Two, ISD practice is, in fact, based on "models" (e.g. the ADDIE model, Keller's ARCS model, etc.). However the word "model" in the ISD domain is used more to mean "something to be imitated", since traditional ID models in fact describe the steps of the development process of instructional strategies, bearing no analogy with the very specific meaning we have been using for a *model* throughout this thesis. So, I once again ask the reader to not let be confused by the "overloaded" meanings of the word model.

necessarily a sound and proven theory backing them. Phenomenological models will be discussed in Section 3.8. The complete lifecycle can be done within minutes and exclusively by a domain expert, as long as, as it will be explained in the next chapter, the chosen target platform description is available.

In practice, the multi-model approach is an essentially *emic* process, i.e. practitioners decide the language to be used themselves while standardized EMLs are inherently *etic*, being an outsider - the designer of the language - the responsible for defining what is important to be represented. Also, I maintain that the Multi-EML approach, by allowing changing the metamodels, models and applications so as to fit specific and contextualized needs, are eminently *idiographic*.

The Multi-EML's emic/idiographic approach, I believe, would then be much more adapted to the following situation, described by (Wilson, 1997):

> The postmodern designer will always feel somewhat ill at ease when "applying" a particular model, even the more progressive models such as cognitive apprenticeship, minimalist training, intentional learning environments, or case- or story-based instruction. Designers should always be playing with models, trying new things out, modifying or adapting methods to suit new circumstances.

The following table summarizes what has been scrutinized in this section:

Table 1: Comparing ISD, EML, and Multi-EML

	etic/emic	***nomothetic/idiographic***	***lifecycle***
ISD	etic	nomothetic	long
EML	etic	idiographic (within a metamodel)	short (model) long(metamodel)
Multi-EML	emic	idiographic	short

It is no surprise that the change from ISD to EMLs and to Multi-EML follows the evolution that software development process has been gone through in the last 30 years or so: from the top-down *waterfall model* to the *iterative and incremental process* and to the today omnipresent *agile methods*, where top-down changed to bottom-up and iteration cycles

reduced from weeks and months down to days. In fact, Gibbons & Rodgers have already traced the analogy of traditional ISD design processes to waterfall-based processes:

> [I]nstructional design is a process approach to design problem solving analogous to the waterfall process found early-on in other design fields but later de-emphasized. (Gibbons & Rogers, 2007)

Also, before concluding this section, I would like to make an analogy between these three approaches for building learning scenarios and the world of toys. In this case the traditional ISD scenarios could be compared to traditional "turnkey" toys, conceived for a specific use. EML-based scenarios, in turn, could be likened to Lego toys, i.e. the famous interlocking plastic bricks that can be connected in different combinations. That is, EML opens the ISD "black-box", allowing designers to have access to its constituent parts and to rearrange them to build different scenarios, just like children snap Lego pieces together to make different figures. Although presenting "universal" bricks, Lego parts are usually bundled in boxes respecting "themes", which provide more specialized pieces and accessories, making them suitable for a more specific use. Of course, in the lack of a suitable part, children always find a way out to finish their assemblies – for example, by equipping their cars with square bricks instead of specialized parts in the form of tires. Likewise, scenario designers can make do with "square tires" when there is not round ones – i.e. the expressive model concepts they need. Finally, Multi-EML could be likened to children having the power of building their own interlocking parts with the shape they want, so as to address their most specific needs.

3.7.4 IMS-LD as an all-encompassing EML

I could not overstate the importance of the IMS-LD specification for the domain of educational modeling languages, especially for popularizing the EML approach among the learning designers and for offering a runtime solution that leverages a metamodel that can be useful in many situations. If I propose now a multi-EML approach, this is, to some extent, consequence of the IMS-LD having put forward a single EML approach.

However, especially when it comes to tackle the problem of expressiveness, I believe that IMS-LD fell short of giving a convenient solution to it. In fact, IMS-LD cannot conveniently express any learning scenario -- contrary to its authors' claim (Tattersall, Burgos, & Koper, 2006). That is, one should not assume that every pedagogical situation can be optimally

represented as chronological sequences of activities controlled by conditional logic. On the contrary, I argue that this is a punctual view optimized for specific process-based scenarios.

Neither can I agree with the claim of IMS-LD being pedagogically neutral (IMS-LD, 2003). The simple fact of the IMS-LD metamodel being used obliges learning scenario designers to think in terms of processes (i.e., the design of a prepared sequence of events, consuming resources required by the activity and producing some outcome), and/or workflows (i.e., a synchronized sequence of tasks performed by roles enriched with conditional logic). This automatically rules-out any possibility of reasoning about how this same scenario could be otherwise interpreted, particularly any interpretation that draws on ideas from outwith rationalism.

In fact, processes and workflows are heavily influenced by the computer architecture, originally proposed by von Neumann, being therefore highly optimized for machines. While the notion of process is much older than von Neuman himself, its current pervasive meaning, at least in the information technology domain, is epitomized by his ideas. Likewise, the concept of a workflow has its control logic dictated by von Neumann's theory:

> Von Neumann called it "conditional control transfer". This idea gave rise to the notion of subroutines, or small blocks of code that could be jumped to in any order, instead of a single set of chronologically ordered steps for the computer to take. The second part of the idea stated that computer code should be able to branch based on logical statements such as IF (expression) THEN, and looped such as with a FOR statement. "Conditional control transfer" gave rise to the idea of "libraries," which are blocks of code that can be reused over and over. (Ferguson, 2004)

If the "Neumann style" is not a consensus, even for programming language specialists (Backus, 1978), it would seem much less likely to be so amongst learning designers. To summarize, if it makes convenient and straight to execute in computers, it loses touch with human natural reasoning.

As stated earlier, we are dealing with potentially different paradigms, what almost always writes off any possibility of having a common representation capable of regrouping concepts from distinct paradigms. Thinking in terms of workflows means *not* thinking in terms of other paradigms. Any attempt of this sort is predestined to failure in representing scenarios stemming from different worldviews.

I used IMS-LD as an example in this section, due to the fact that it is the most widespread and also the one considered to become a standard. However, the same applies to any other EML intended to model any educational experiment, regardless of paradigms. As another example, I can point out the Learning Design Language (Ferraris et al., 2005), which, on top of that, also singles out the activity-centered approach decried earlier here and elsewhere (Allert, 2004) – even though with particular inflections (Martel, Vignollet, Ferraris, David, & Lejeune, 2006). In sum, no single language can be offered as a panacea for all problems of representation of learning scenarios.

3.8 A philosophical account for the multi-model approach

Believing that the logical positivism is no longer tenable in modern science, philosophers have proposed ideas that soon found echo in other areas, like science, education, art, etc. Post-modernism, post-positivism, post-structuralism, critical multiplism are some examples of terms that reflect these ideas and that share the common belief in the impossibility of viewing life from an objective stance.

Thus, whereas modernists are in a sempiternal truth-seeking enterprise, postmodernists deny the existence of an absolute and idealized view of truth and *"replace it with a dynamic, changing truth bounded by time, space, and perspective"* (Wilson, 1997); while modernists, in order to explain the truths they believe, advocate integrative all-encompassing theories and ideals, post-modernist movements deliberately dismiss them as unfit for a dynamic and decentralized modern life, and replace the modernist's so-called *Big Stories*, or meta-narratives, with "*petits récits*" (Lyotard, 1979). In other words, post-modernists consider that the "Western scientific thought is flawed with its emphasis on sameness" (I. Visscher-Voerman & Gustafson, 2004).

> It is my contention that the multi-model approach provides an ontological answer to the post-modernism's relativism.

Furthermore, (Wilson, 1997) considers that the most manifest expressions of post-modernist ideas into the educational area can be found in the constructivism, giving rise to such a symbiosis that *"there may be some confusion as to how postmodernism is different from*

constructivism". Thus, I argue that the multi-model approach offers a more amenable environment in which constructivist scenarios can be built.

Finally, to wrap up this section, I would like to stress one of the guiding principles of this memoir:

> The objective of this dissertation is not to criticize, adopt or recommend any position – modernist or post-modernist -, but to propose a technical framework that accommodates indistinctly both worldviews.

3.8.1 Multi-EML and Phenomenological models: Designing learning scenarios as an extension of the user experience

Phenomenology is a branch of philosophy that studies phenomena, i.e., things as they are *perceived*, as opposed to the study of things as they *are*. Although the concept of phenomenology was developed by different philosophers, this section is more in line with the meaning of it as used by some scientific disciplines, which seems convenient enough for our purposes. Another approach would be to draw on the *existential phenomenology*, proposed by Heidegger[17], and attempted in the domain of computing science by (Winograd & Flores, 1986), but not without some criticism (Hollingsworth, 2005).

It is important to note that the concept of phenomenology has already been appropriated by different disciplines, like Chemistry, Physics, Biology, Economics, etc. for their own purposes. For example, the Encyclopaedic Dictionary of Physics states that "a phenomenological theory relates observed phenomena by postulating certain equations but does not inquire too deeply into their fundamental significance". To (Cartwright, 1983), there is a distinction between phenomenological and theoretical laws, in that "phenomenological laws are about appearances; theoretical ones are about the reality behind the appearances".

[17] To Heidegger, the scientific analysis is just a founded mode, a particular type of investigation which is "non-primordial". The phenomenological analysis, while reacting against the science's tendency "to posit some thing 'behind' the perception", and based on a much more "primordial" foundation of practical, everyday knowledge is considered a by him more fundamental than science itself.

Applying the above ideas to the modeling field, will give rise to the so-called *phenomenological models*. An interpretation, attributed to (McMullin, 1968), defines phenomenological models as models that are **independent of theories**[18]. Some authors consider that this *theory-independent* view of models seems just too strong:

> Phenomenological models, while failing to be derivable from a theory, incorporate principles and laws associated with theories […] though usually not the complete theory (Frigg & Hartmann, 2006).

Other authors, though, seem to agree verbatim with this idea. (Cartwright, 1999), for example, goes even further, by advocating that theoretical (top-down) models are not realistic, while empirically observed (bottom-up) models are.

Cartwright's criticisms are founded on the fact that theories and their formulae usually assume ideal conditions - e.g. infinitesimal dimensions, frictionless surfaces, perfectly transparent glasses, ideally round bodies, and so on -, which do not exist in real situations. For example, Isaac Newton's famous law of gravitation is an ideal law relating the force exerted by two masses at a distance valid only for particle-like objects. Therefore, this law can only be true under very simple circumstances.

This interpretation calls for models designed as quick and draft conceptualizations reflecting very circumstantial needs, and far from ideal conditions, drawing on phenomenological rather than metaphysical criteria. As Ernan McMullin (*apud* Koperski, 2006) explains, "sometimes physicists—and other scientists presumably—simply want a function that summarizes their observations. Curve-fitting and phenomenological laws do just that".

Additionally, when creating models, designers should also be guided by their *feelings* about what works and what doesn't, based on their experience, "on the assumption that feelings are 'had' or experienced before they are 'known'" (Dewey, 1934 – *apud* (Howe, 2006)).

Thus, I sustain that the Multi-EML approach, as proposed here, can be used by designers wishing to create phenomenological models. That is, the Multi-EML approach comes to the

---

[18] In fact this is not the only interpretation to phenomenological models. Another common understanding is that models are said to be phenomenological only insofar as they describe the behavior or properties of empirically observable systems, refraining from describing "hidden mechanisms and the like".

rescue of designers who are more interested in leveraging their experience, observations and feeling than in confirming first principles from overarching theories. Thus, by enabling designers to materialize their broad insights, many times appeared on the spur of the moment, the Multi-EML approach can make of the design activity a direct extension of the practitioner's experience.

3.9 Conclusion

This chapter intended to motivate the need for a multi-model approach, called Multi-EML for the domain of learning design, for representing learning scenarios. Based on a few characteristics presented by good models – e.g. expressiveness and specificity - I set out to justify an approach that preserves the interests and values of a scenario by offering a means of modeling different interpretations for it, possibly drawing on distinct paradigms.

This approach contrasts with the purely *hierarchical conceptualization* afforded by single models - which may be suitable for simple enough domains, but which falls short of delivering expressiveness to designers working in richly multi-dimensional domains. More specifically, for the domain of learning design –certainly one of these rich domains - I argued that the "one-dimensional" approach, epitomized by the standardized one-fits-all language advocated by the sponsors of the IMS-LD specification, is not sufficiently expressive to represent the different learning strategies and at the same time to preserve their original intentions.

(J. Coplien, 2000) had already discussed the question of addressing complex domains with single models:

> Neither nature nor business has been so kind as to formulate problems with intrinsic hierarchical structure.

Like Coplien's Multi-Paradigm Design, Multi-EML approach addresses complexity with sets of conceptualizations, aiming at different concerns, which taken together present more of a *network-like* structure than a hierarchical one.

Also, like other approaches for the design of learning scenario – e.g. IMS-LD and ISD - there is no way of guaranteeing that the desired learning outcomes will be attained. However, being crafted by the domain experts, the metamodels defining the scenario's language can be

regularly revisited as needed, and, through successive interactions, can gradually shorten the gap between instructors' expectations and actual educational outcomes.

Next chapter I will describe a technology, called Language Oriented Programming, which rises to the challenges envisaged for the implementation of the Multi-EML approach.

Chapter Four: Language Oriented Programming: going from languages to programs

4.1 Introduction

In the preceding chapter I described the Multi-EML, a design approach that suggested the creation of multiple models to afford complementary insights of leaning phenomena, on the grounds that it would leverage the expressivity of designed models, particularly in the richly multi-dimensional domain of education. I also suggested that one way of how this could be achieved was by enabling "downstream designers" – a role that should ideally be fulfilled by domain experts - to take part in the design, by offering them a way in which they can *easily* conceive their own design languages, with the relevant semantics and at the level of expressivity and specificity that suit their needs.

I highlighted the adjective "easily" in the above paragraph because it is important to note that I am not talking about simply offering a means of creating a language in any possible way, since there are already several off-the-shelf applications allowing this and whose manipulation can be challenging for non-programmers. For example, "easy" rules out asking domain experts to use, say, the XMLSpy editor for creating an XML Schema file representing a given conceptual model, just because this solution is *not* easy – at least for the clientele I envisage for the Multi-EML approach, e.g. experts in the domain of education.

Furthermore, even if XML Schema files could be easily authored, by itself they are not incredibly useful, and require other artifacts to leverage their use that are even more complicate to produce, such as APIs, editors, "players", etc. In fact developing moderately complex such artifacts may require a familiarity with a wide range of technologies for describing software and the savvy to make them work together without losing focus on the user requirements:

> Although we increasingly use different languages for different tasks - programming languages for writing application logic, XML for transmission of data between application components, SQL for storing and retrieving data in

> databases, WSDL for describing the interfaces to web-facing components – there are many complexities involved in getting these languages to work effectively together, and none of them directly address the business problem faced by the end user (Cook, 2004).

This takes us to the next point to be observed by Multi-EML implementations: constructing multiple languages is one matter; having tools to express the abstractions of these languages is quite another. That is, enabling the construction of multiple languages dovetailed to the needs of designers is only halfway through the whole process, which also encompasses producing artifacts that realize these languages and that can be used by end users.

To summarize, the Multi-EML approach needs a technology that offers two essential features:

- Allows the easy creation of languages describing learning scenarios.
- Allows the easy creation of applications reflecting the aforementioned languages.

This chapter intends, then, to shed some light onto the technology that will help us both to build design languages and to develop full-fledged applications reflecting these languages, possibly involving all the complexity above described, but *requiring the least of computer related expertise* – ideally, the whole development cycle should be within the reach of the domain experts.

Since this thesis has a particular interest for the domain of learning design, an envisaged scenario is a learning designer or instructor -- not necessarily computer-savvy -- wishing to "transform" his or her *know-how* about, say, a specific teaching strategy into an executable application. Thus, the candidate technology should allow him/her to represent the know-how's conceptualization in an informal notation (e.g. natural language) and then to transform this notation into executable code without requiring any - conventional - programming skill, all the while keeping the whole development process to a reduced time frame, so the designer can promptly see his/her conceptualizations "in action". Also, any change operated on the conceptualization should cause an immediate and equivalent change in the final application.

This apparent programming nirvana is, for the moment, still not fully attainable, but there are already quite a few indications that this objective will be in a not so distant time achieved by a few related – and competing - technologies, proposed by some of the key world players in the software industry and that I will present in Section 4.7. In any of these technologies, an

intermediate semi-formal notation for the domain conceptualization is required to move from the initial informal language to the final computer code. Thus, the first part of this chapter (namely, Section 4.2 and 4.3) discusses how these semi-formal specifications can be obtained.

4.1.1 LOP and Domain Specific Languages

Part of the originality of this chapter resides exactly in the fact that, instead of simply choosing any suitable technology and implementing a solution for the Multi-EML approach, I decided to step back and investigate what all the candidate technologies have in common, bringing this discussion to a "meta" level, in an effort to apprehend the essence of these technologies.

Discussing at a "meta" level usually means to find a common name whenever we want to refer to a collection of related things as a single category. So, the first step is to find a term or expression to designate these different but related technologies. As I will discuss more fully in Section 4.4, I decided to adopt a designation proposed by Martin Fowler, which will spare us from writing "these technologies" every now and then: **Language Oriented Programming** or simply **LOP**. I made this choice because this expression is fortunate enough to capture the quintessence of these technologies: before all, they offer ways of working at the level of (domain) languages. This expression subsumes technologies like Model Driven Architecture (MDA), Generative Programming, Intentional Programming, Meta-programming, etc., perhaps more popular, but that seem particular cases of *language programming* (e.g. a language is more generic than a *model* or language programming is not only about *generating* code, etc.)

To the best of my knowledge, there is no single work in which different existing LOP implementations are compiled and compared. Related theses and papers limited themselves to compare the particular instance of the LOP approach in study – for example, MDA - with other more conventional programming paradigms – e.g. object oriented. The existing comparative literature on the different LOPs is provided by the LOP implementation's own authors, who sometimes cross-reference one another, to provide examples of related work, but who do not offer a broader picture including all players. So, to make a choice for one particular implementation, I had to make the comparison myself and this chapter surfaces a part of this effort.

Fundamentally, the existing LOP variations are extensions of the idea that the hardest part in the design of a computer program is the conceptualization behind it and that everything else is rote task that should be spared from programmers. As (Liddle, 1996) puts it:

> Software design is the act of determining the user's experience with a piece of software. It has nothing to do with how the code works inside, or how big or small the code is. The designer's task is to specify completely and unambiguously the user's whole experience....The most important thing to design properly is the user's conceptual model. Everything else should be subordinated to making that model clear, obvious, and substantial.

Certainly, such an ambitious venture can only be feasible if the problem space is confined to a specific domain with agreed upon concepts and clearly defined rules. So, before proceeding to the main objective of the chapter, namely describing the LOP different implementations, I will provide some background matter of the LOP foundation and discuss how they can help us constrain problems. Most of these subjects revolve around the Domain Specific Languages, or DSLs - design languages formal enough so as to be realized in computers – and their related concepts. This is not a capricious choice, but results from DSL being a mature and proven subject, widely used in Computer Science.

Also, from the beginning of this dissertation I have been using the term *model* roughly connoting conceptualizations from a specific domain, but without having it properly defined. Thus, I will seize the opportunity to redefine in this chapter the concept of *model* and many other associated concepts, by relating them to Domain Specific Languages (DSLs), which are themselves couched in a more rigorous footing.

Finally, even if the main objective of the chapter is to set forth the state of the art of a subject that deals with the design of computer programs at large, whenever appropriate, I will re-cast the discussion in terms of the learning design domain, establishing relevant links with previous chapters.

4.2 Domain Specific Languages (DSLs)

The history of programming languages is a steady struggle towards making them closer to the way human think. From first-generation (i.e. machine) languages, with their 0’s and 1’s, to second-generation (assembly) languages, with their mnemonics, and to modern third-

generation *general purpose languages* (GPL), which permit to elaborate programs using constructs taken from natural languages –usually the English –, programming languages have made a great deal of progress in moving away from the machine.

4.2.1 Defining DSLs

Another way of saying that languages move "away from the machine", is affirming that they present increasing levels of abstraction, resulting from layers of innovations, each one stacking on top of its predecessors:

> Every generation of programmers uses its era's programming languages and tools to build the programs of the next generation. Layers of abstraction have accumulated like geological strata (Rosemberg, 2007).

Furthermore, as they move up the abstraction strata, languages tend to acquire some degree of specialization, which make them more adapted to some applications. This has been happening in the computer domain since early ages with, for example, Cobol for business applications and Fortran for scientific and engineering calculations. In fact, the need to solve problems from particular domains has always been placed on the top of the agenda of computer programmers, who developed over time different solutions (van Deursen, Klint, & Visser, 2000):

- *Subroutine libraries:* the oldest method for "packaging reusable domain-knowledge".
- *Object-oriented frameworks* and *component frameworks:* while subroutine libraries contribute domain specific code to be invoked by the application, now it is the framework that has the control and that invokes application-specific code contributed by the programmer. For this reason some call frameworks *upside down libraries*.
- *Domain-specific languages* (DSLs).

While I am not interested in studying libraries or frameworks in this thesis, I will focus on Domain Specific Languages (DSL). One of the most frequently cited definition for a Domain Specific Language can be found in (van Deursen et al., 2000):

> A domain-specific language (DSL) is a programming or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

So as to clarify and further explore the many ideas contained in this definition, I decided to break it down, commenting on its main points. So, we get down to the following:

Programming or executable specification language: Considering the declarative nature of DSLs, at first sight, "executable specification" sounds like an oxymoron, since, analyzing from the point of view of the past and present mainstream programming paradigms – e.g. procedural, object oriented, etc. -, declarative specifications are usually not executable[19]. That is, while declarative specifications usually subsume a list of requirements expressed in a formal or informal notation being usually intended for human consumption, executables are essentially meant for machine consumption[20]. Thus, doing away with this barrier separating a declarative specification from execution is one of the distinguishing features of DSLs.

Notations and abstractions are two rather informal terms to connote, respectively, the concrete syntax and the abstract syntax of a language. In other words, while the latter basically represent (abstract) concepts, the former is the way these concepts can be communicated to human and machines. In the next section concrete and abstract syntaxes are covered in more detail. The idea of **expressive power**, or expressiveness, which has already been discussed at length in Chapter Three, reflects the idea of a language possessing the "right words" to represent the existing concepts of a domain.

As for a **problem domain,** if we really want to understand what the authors meant by this expression, we have to search somewhere else, since (van Deursen et al., 2000) hedge any categorical definition:

> Our definition of Domain Specific Languages inherits the vagueness of one of its defining terms: problem domain. Rather than attempting to define this volatile notion as well, we list and categorize a number of domains for which DSLs have actually been built.

---

[19] A notable exception to this practice is found in the database domain, where a declarative language (SQL) is executed (interpreted) at runtime.

[20] Contrasting to "declarative specifications" there are "imperative specifications", which involve programming a system via sequential, procedural instructions, typical of third generation languages (3GL). This kind of specification is usually executable (after compilation).

Nevertheless, (Krzysztof Czarnecki & Eisenecker, 2000), p. 34), taking a bolder stance, define a **domain** as "*an area of knowledge"* that:

- is scoped to maximize the satisfaction of the requirements of its stakeholders
- includes a set of concepts and terminology understood by practitioners in that area; and
- includes the knowledge of how to build software systems (or parts of software systems) in that area

4.2.2 Abstract syntax, concrete syntax and semantics of DSLs

DSLs, like any language, include an *abstract syntax*, a *concrete syntax* and *semantics*, concepts that need to be clarified in this chapter. Nevertheless, the idea here is just to provide enough guidance for the purpose of facilitating the comprehension of this text and not to treat them in rigorous mathematical footing, what would be beyond the scope of this thesis.

4.2.2.1 Abstract syntax

To start with, **abstract syntax** represents the body of concepts in some area of interest as well as the relationships between them:

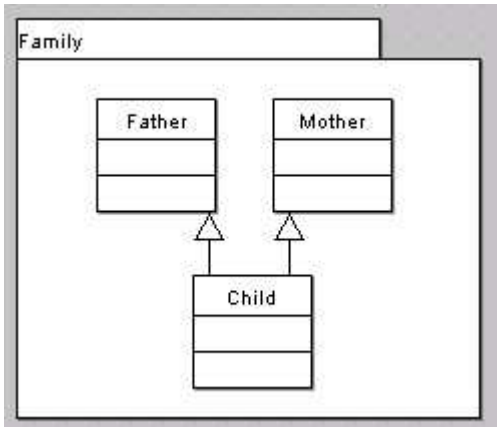
> The abstract syntax of a language characterizes, in an abstract form, the kinds of elements that make up the language, and the rules for how those elements may be combined (Cook & Kent, 2004).

Abstract syntaxes do not depend on data structures or encodings, as they are at the level of ideas. As with any artifact, they are purposeful and represent, for someone, a simplified view of the domain under consideration.

4.2.2.2 Concrete syntax

Abstract syntaxes are mental conceptualizations and, as such, can be useless if we don't find a convenient notation allowing it to be communicated to other people or to machines. That is why we need to designate symbols to each abstract concept, so that they can be codified into some physical property that can be "sensed" by humans or computers – and further reasoned about or processed. The collection of all these symbols constitutes the **concrete syntax** of a language. Figure 16 illustrates three different concrete syntaxes referring to the same abstract

syntax - (a) is a typical UML diagram, (b) is a tree-like notation and (c) is a textual notation (XML).



(a)

(b)

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="family"
    nsPrefix="">
  <eClassifiers xsi:type="ecore:EClass" name="father"/>
  <eClassifiers xsi:type="ecore:EClass" name="Mother"/>
  <eClassifiers xsi:type="ecore:EClass" name="Child" eSuperTypes="#//father #//Mother"/>
</ecore:EPackage>
```

(c)

Figure 16: Examples of concrete syntaxes: (a) UML-like (b) tree-like and (c) textual (XML)

4.2.2.3 Relating abstract syntax and concrete syntax

The relation between abstract syntax, concrete syntax and the entity they represent has been long illustrated by the Triangle of Ogden and Richards (or semiotic triangle), reproduced in Figure 17.

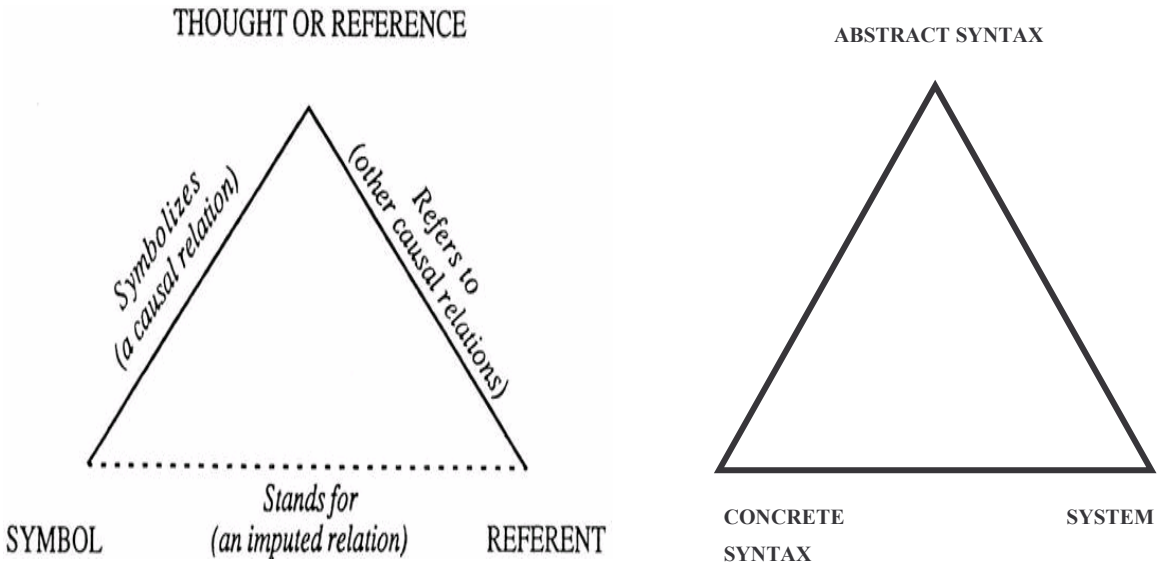


Figure 17: The Triangle of Ogden-Richards as original (left) and adapted (right)

Internally, computers usually represent both abstract and concrete syntaxes as tree structures (i.e. graphs in which any two vertices are connected by exactly one path). In fact, even when writing texts, programmers are always reminded of this type of structure, e.g. whenever they are obliged to use brackets, braces, parentheses, etc. in their programs. A *parser* is then used to transform the programmer's usually textual editions into Parser Trees (or Concrete Syntax Trees) and subsequently into Abstract Syntax Trees, or AST.

Parser trees differ from ASTs in which they contain syntactic information such as layout (white space and comments), parentheses, and extra nodes that are not germane for the semantics of the program. Using Peter Landin's terms, Concrete Syntax Trees present **syntactic sugar** – i.e. branches and leaves that add nothing to a program's expressiveness, serving only to make it "sweeter" for humans to understand – which are removed in the streamlined AST. In further steps, ASTs are transformed by a parser into primitive instructions and data structures, which are then executed.

In general, concrete syntaxes for imperative languages are textual, while concrete syntaxes for declarative languages combine graphics, text, and even conventions used by some tools in the human machine interface. A common way of describing concrete syntaxes is by using

context-free grammars[21]. (Kleppe, Warmer, & Bast, 2003) explains us that the Bakus Naur Form (BNF) has been historically used to define context-free grammars:

> In the past, languages were often defined using a grammar in Backus Naur Form (BNF), which describes what series of tokens is a correct expression in a language. This method is suitable and heavily used for text-based languages, like programming languages. We could use a BNF grammar to define modeling languages. It does fulfill the requirement that it is suitable for automated interpretation. However, BNF restricts us to languages that are purely text based.

For example, a simple context-free grammar could be created using the BNF form that defines well formed expressions involving the basic mathematical operations (Listing 1):

Listing 1: BNF example

```
N=[expr, op, var]
T=["+","-","/","*","x","(",")"]
S = <expr>
```

And *R* could be represented as:

```
(1) <expr> ::= <expr> <op> <expr>
             | ( <expr> <op> <expr> )
             | <var>

(2) <op> ::= "+" | "-" | "/" | "*"

(3) <var> ::= x (x = 0, 1, 2, ...)
```

4.2.2.4 Semantics

Having a syntactically well-formed sentence is not enough. In order to be useful, sentences must make sense for someone in some context. In his now famous sentence "*Colorless green ideas sleep furiously*", (Chomsky, 1957) gave an example of a language construction whose syntax is flawless, while being completely nonsensical.

---

[21] According to (Aaby, 2004), context-free grammar is a quadruple (*N,T,S,R*) where,

- *N* and *T* are disjoint sets, called **terminals** and **non-terminals** (symbols), respectively;
- *S* is a non-terminal, the **start category**;
- R is a finite set of **production rules** of the form N → V; where V is a string of terminals and/or non-terminals.

Compared to syntax, **semantics** is a much less advanced domain, as far as automation is concerned. As (Simon, 1996) explains,

> The linguistic theory has thus far been largely a theory of syntax, of grammar. In practical application to such tasks as automatic translation, it has encountered difficulties when translation depended on more than syntactical cues – when it depended on context and meaning.

In computer science there are two practical problems concerning semantics: one is to translate natural languages into computational constructs and the other is to formally define the semantics of these computational constructs (e.g., DSLs and general purpose programming languages) with mathematical rigor. While the former problem is still a long way from providing satisfactory results, the latter was brought relatively under control with some techniques, described hereafter.

We are interested here in the problem of the semantics of computer languages are defined, particularly for the case of DSLs. Among the most used techniques, one that presents particular practical relevance to the automated processing of domain specific languages is the *Translational Semantics*. It basically means that expressions in a given DSL will be translated into another language that already has its semantics well established. For example, our DSL to describe a certain type of pedagogical scenario can have its semantics defined by translation to, say, the Java programming language. The Java code, in turn, can have its semantics translated (more precisely, compiled) into Java bytecodes, which will once again make use of translational semantics techniques to arrive at the instruction set of a particular CPU platform.

However, it is fair to think that we are just pushing the problem to a lower level (i.e. where has the CPU's instruction set been defined, in the first place?). That is when formal semantics comes into play. In computer science, formal semantics is an expression that subsumes three broad classes of techniques, called *denotational* semantics, *operational* semantics and *axiomatic* semantics. They all rely on the premise that the most precise language we have is that of mathematics (Krishnamurthi, 2003) and, hence, find a way into a mathematical model that describes the possible computations described by the language. That is, when it comes to computer-based languages, semantics comes down to explaining how interpreters (operational semantics) and compilers (denotational semantics) for these languages work; and since we are

imbued with the task of *explaining* something, we can only to do it in terms of a language. That's where mathematics comes in, by offering a universal and formal language.

Having explained DSLs and some basic concepts like syntax and semantics, we are now in a better position to discuss other related concepts, like model - which has been only loosely introduced in Chapter Two – and metamodel, by establishing a relation between them, which is what I will do in Section 4.3. But before that, I will explain how DSLs are designed and cite some advantages and disadvantages in the use of DSLs.

4.2.3 Designing DSLs

The learning scenarios described in preceding chapters will have to be somehow designed. To employ the concepts seen in this chapter, for each scenario, one or more DSLs will be created that reflect the vocabulary needed to express the knowledge of domain experts. Please note that this section can be seen as the computer-based counterpart of Section 2.3.1, when we discussed the creation of Design Languages, using the Peer Feedback language as an example.

To (van Deursen et al., 2000), the development of a domain-specific language typically involves the following steps:

- **Analysis** (1) Identify the problem domain. (2) Gather all relevant knowledge in this domain. (3) Cluster this knowledge in a handful of semantic notions and operations on them. (4) Design a DSL that concisely describes applications in the domain.
- **Implementation** (5) Construct a library that implements the semantic notions. (6) Design and implement a compiler that translates DSL programs to a sequence of library calls.
- **Use** (7) Write DSL programs for all desired applications and compile them.

According to (M. Fowler, 2005), there are three main parts to defining a new DSL:

- Define the abstract syntax, that is, the **schema** of the abstract representation.
- Define a **generator**. This describes how to translate the abstract representation into an executable representation. In practice the generator basically performs a transformation from the (domain specific) source language to the (general purpose) target language and, in so doing, defines the semantics of the DSL.

- Define an **editor** to let people manipulate the abstract representation through a projection, so that users may be able to easily create DSL programs.

Compared to the precedent listing, Fowler's emphasizes, *en gros,* the need of providing an *ad hoc* editor for the DSL. He calls attention for the fact that, more than just an implementation matter, offering flexible, user-friendly editors can be a key factor in engaging the domain expert in the development efforts.

After the analysis phase, a DSL derived from the Peer Feedback language illustrated in Section 2.3.1, could look like the depiction shown in Figure 18 (note the arbitrary choice of both abstract and concrete syntax):

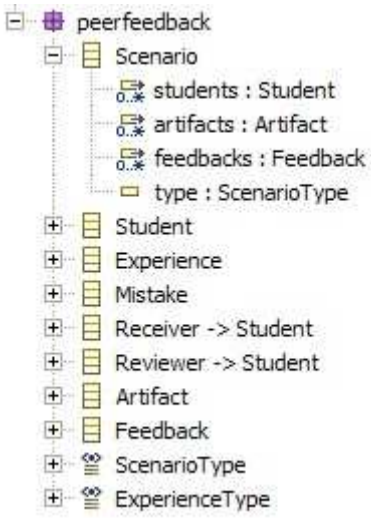


Figure 18: A Peer Feedback DSL

It is important to note, also, that most DSLs aren't actually *designed*, but *developed* in a piecemeal growth that may span for as long as they are used. Should this happen, the above steps would be constantly revisited. For the example, the Peer Feedback DSL could be constantly "under development", yielding new *entities*, *rules of grammar* and *actions,* possibly differing from the ones shown in Figure 18.

4.2.4 Advantages and disadvantages of using DSLs

The main advantage of a domain specific language is that you have types, syntax and operations that map directly to the concepts in your domain. However, many others will follow. According to (van Deursen et al., 2000) DSLs are beneficial because[22]:

- DSLs make use of abstraction from the problem domain. So, domain experts can understand them and consequently take part in the development effort (i.e. validate, modify, and often even develop) of DSL-based programs.
- DSL programs are concise, self-documenting to a large extent, and reusable
- DSLs enhance productivity, reliability, maintainability, and portability.
- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge.
- DSLs allow validation and optimization at the domain level.
- DSLs improve testability.

Some of the *disadvantages* of the use of a DSL, also according to van Deursen, Klint and Visser (*ibidem*), are:

- The costs of designing, implementing and maintaining a DSL.
- The costs of education for DSL users. That is, DSLs are supposed to save us time when working, but they don't spare us from having to learn them.
- The limited availability of DSLs.
- The difficulty of finding the proper scope for a DSL.
- The difficulty of balancing between domain-specificity and general-purpose programming language constructs.
- The potential loss of efficiency when compared with hand-coded software.

(M. Fowler, 2005) adds that the fundamental issue in using DSLs is the trade-off between the benefits of using them versus the cost of building the necessary tools to support them effectively, an idea that (Simonyi, 1996) formulates in a more metaphorical way:

---

[22] Please note that (van Deursen, Klint and Visser, 2000) is in fact a bibliographic compilation of the area of DSL and, as such, provides an extensive list of references to some of the most important works in the domain. Consequently, the advantages and disadvantages here listed were collected from different sources and the reader may refer to them to have more detailed information.

> Our human civilization rests on the power of abstraction insofar as the volume of specific cases handled by an abstraction is typically much greater than the overhead of the abstraction mechanism itself.

Of special importance to this dissertation is the DSL's capacity to enable domain experts to take part in the development effort in their domain. And this is only possible if the necessary applications can be created without requiring programming skills. I am going to revisit this particular point in Section 4.5.

4.3 Models and Metamodels

Chapter Two has introduced the concept of a *model* - a design artifact referring to only some aspects of an entity and serving a purpose. But I did not provide then any formal enough definition so that models could be considered for computational tasks.

I then begin by providing the definition for a metamodel that will be used in this chapter:

> Metamodeling has come to mean the construction of an object-oriented model of the abstract syntax of a language. […] A metamodel characterizes language elements as classes, and relationships between them using attributes and associations (Cook & Kent, 2004).

What does make a metamodel different from a DSL, then? The first distinction is explicit in this definition: it is a language couched in terms of object oriented constructs (e.g. class, attribute, association, etc)[23]. This definition discloses the fact that a metamodel is a concept intimately related to the object oriented programming paradigm and that has been further popularized by the Unified Modeling Language (Booch et al., 1999) standard. An excerpt of the UML's abstract syntax is presented in the following diagram, as an example of a metamodel:

[23] I could have adopted a looser definition for *metamodel*, like "the collection of concepts of a domain", as it is common in the literature. But in that case, a metamodel would only stack up to the already large number of different modeling approaches, like entity-relationship model, concept mapping, mind maps, schemas, ontologies, etc., offering no added value.

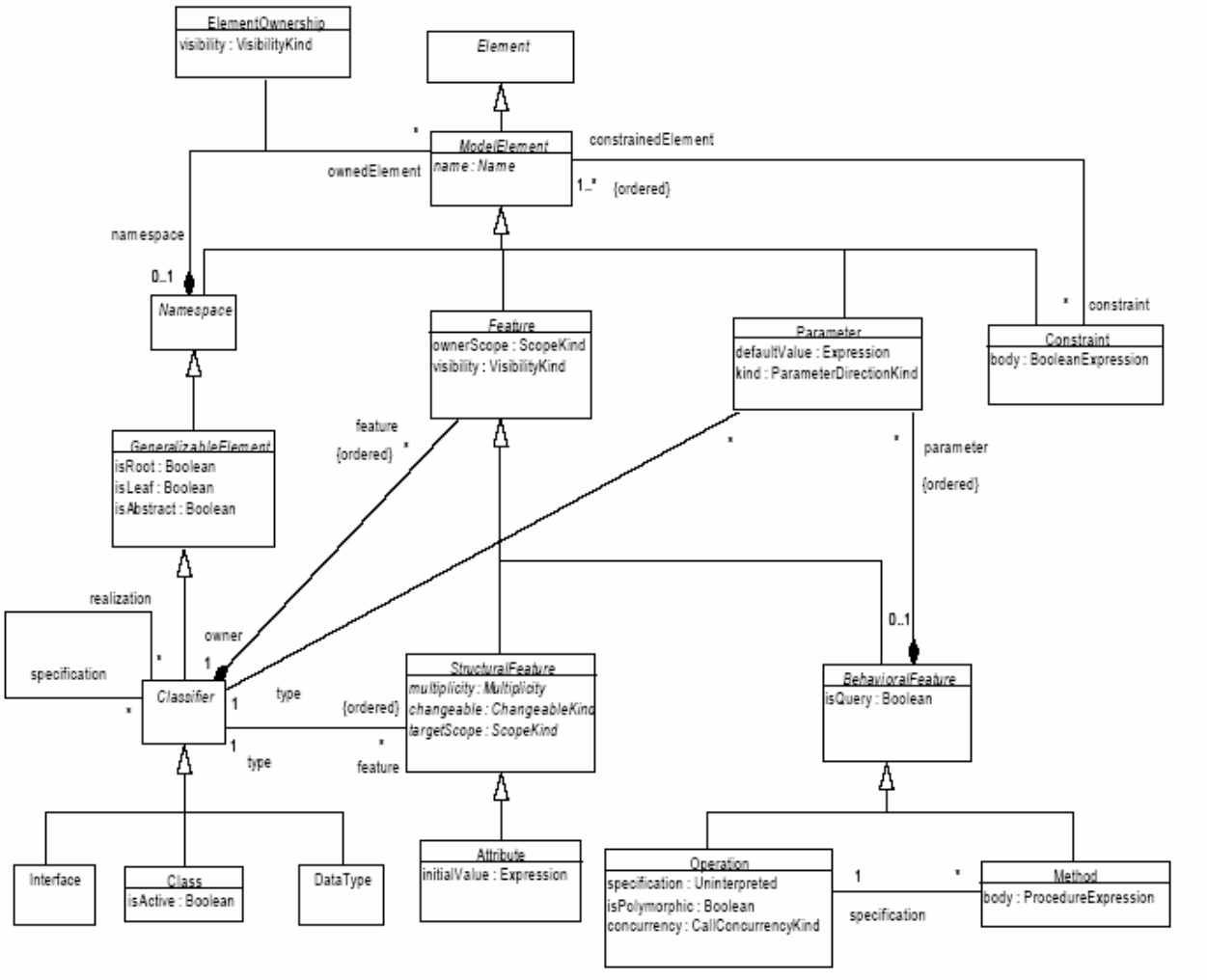


Figure 19: The abstract syntax of the UML standard (Core Package – Version 1.2)

Another distinguishing feature from metamodels (setting them apart from plain DSLs) is that they are not exclusively text-based, resorting frequently to graphical notation, a characteristic that has been extensively used by the UML itself to present its own abstract syntax, as shown above.

Having a definition for metamodel, a model could be defined, *tout court*, as an *instance* of a metamodel. However, given the importance of this concept for this thesis, I will elaborate more on it, enriching this discussion with definitions by different authors.

In this thesis I am particularly interested in the understanding of a model in computer science, so I will focus on definitions from this field. I reproduce four of them:

> A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language.
>
> (MDA, 2003)

> [A model is] a description of (part of) a system written in a well-defined language.
>
> (Kleppe et al., 2003)

> A model is a set of statements about some system under study
>
> (Seidewitz, 2003)

> A model is an abstract description of software that hides information about some aspects of the software to present a simpler description of others.
>
> (Greenfield & Short, 2004)

In each definition above there is a reference, implicit or explicit, to the word *language*. Terms like *statement* or *description* only expose the close relation between models and languages. To summarize, models are representational artifacts whose elements are combined respecting a *grammar* and represented in a specific *notation*, or *concrete syntax.* I'll keep the word model for the *instances* conforming to grammars defined in DSLs or metamodels[24].

Finally, while acknowledging the similarities between models, metamodels and languages (more specifically Domain Specific Languages), I will use in this thesis the terms model and

[24] Without wanting to be too recursive, a metamodel is itself a model whose own metamodel, as explained before, is defined in terms of object oriented constructs. If the distinction between a model and a metamodel seems confusing by now, this is because it really is - even for experts of the domain. As Seidewitz (2003) points out, "There has been, and continues to be, a great deal of discussion within the software community on modeling and metamodeling and the relationships between modeling languages and metamodeling languages. Such relationships' circular nature makes them particularly hard to discuss clearly".

language for generic purposes and conserve the term metamodel for whenever I need to stress the object oriented aspect of a give representation.

4.4 The Language Oriented Programming Paradigm

One of the characteristics of working with DSLs is that programmers are led to think in an abstract way, i.e. a sort of "meta-programming" in which conventional programs are only instances of more generic specifications. Luckily, abstracting is something human beings are naturally good at – in fact, whenever someone says "this is *a kind of*" or "this is *like* that", etc. he or she is exercising the capacity of abstraction.

However, abstracting domains and documenting the abstraction in representational artifacts is not enough – certainly few end users would know how to use a model to solve everyday tasks in their jobs. To be really useful, these representational artifacts should somehow "drive" the automatic generation of other assets, directly manipulated by end users. In fact, these representational artifacts are already being considered for a much more important role by some closely related programming techniques, which are equally characterized by an effort from the part of programmers to eschew from solving problems using general-purpose programming languages. Instead, they opt for, firstly, creating one or more very specific languages adapted only for the problem domain in question (i.e. DSLs) and finally representing and solving the problem in terms of these new languages.

Although each of these programming techniques has its subtleties and idiosyncrasies, they all share the same approach of adding an additional layer on top of existing *general purpose languages* (GPL) so as to bring them closer to the way people -- or at least, domain experts -- think and the same ambition of becoming the new mainstream programming paradigm. To contrast this approach with the present mainstream programming paradigm, the Object Oriented Programming, (M. Fowler, 2005) proposed, as already stated, the term **Language Oriented Programming (LOP)**:

> I use **Language Oriented Programming** to mean the general style of development which operates about the idea of building software around a set of domain specific languages.

This expression is not new, though, and has its root traced back to works from (Dmitriev, 2004) and even earlier from (Ward, 1994). Hence, this is the expression I will adopt in this thesis whenever I want to refer to the whole collection of these (slightly) different techniques as well as to their implementations - though (Fowler, *ibidem*) uses the expression *Language Workbenches* when referring to the implementations. Some of LOP's most popular implementations are OMG's Model Driven Architecture, Microsoft's Software Factories, Czarnecki's Generative Programming and others that I will discuss in section 4.7.

It is indisputable that the productivity of software developers has increased by raising the level of abstraction from assembly languages to third generation programming languages, like Java or C#. Likewise, the proponents of the Language Oriented Programming bet on the gain of productivity to be achieved by raising the level of abstractions to even higher strata.

It is important, at this time, not to lose focus on the question to be answered in this chapter, namely how programming can be significantly simplified so learning designers or even domain experts will be able to contribute to the software development efforts. The next section gives a clue on how this can be achieved.

4.4.1 LOP programming versus GPL programming

According to Dmitriev (ibidem), programming with general purpose languages requires that:

1. Firstly, we create a conceptual model reflecting the problem we want to solve. This is a basically mental activity;
2. Secondly, we select a (generic) programming language;
3. Finally we do the -- usually difficult -- mapping of our conceptual model into a generic programming language (e.g. Java, C++, etc).

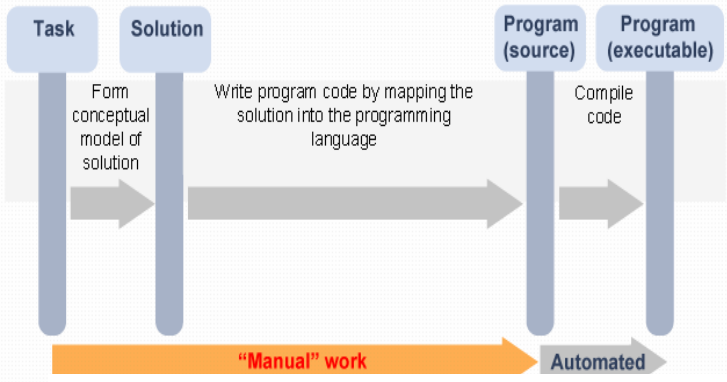


Figure 20: programming with GPLs (source: Dmitriev, 2004)

This last step above is oftentimes the "bottleneck" of programming with generic programming languages, because they are too distant from the way human think, making the mapping *"not easy or natural in most cases",* and consequently demanding professional developers experts in the target generic language.

Contrasting to the programming with GPLs, programmers using LOP-based techniques proceed as follows (Figure 21):

1. This first step remains unchanged in relation to GPL programming, that is, we elaborate a conceptual model of the problem.
2. Instead of choosing a generic programming language, now you select a DSL that better matches your problem -- or create a new DSL if no appropriate DSL exists.

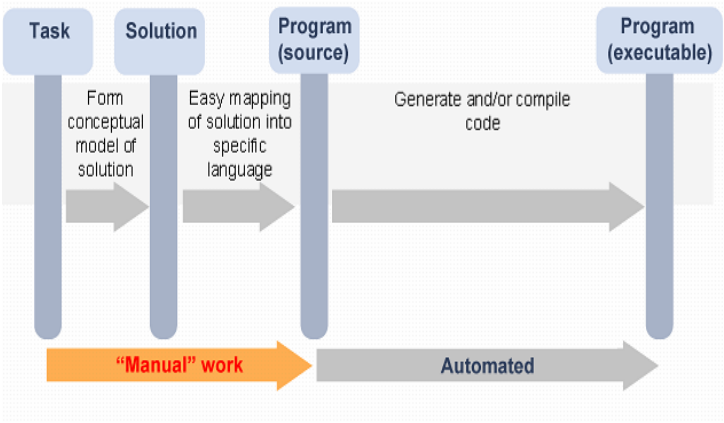


Figure 21: programming with DSLs (source: Dmitriev, 2004)

3. Next you write the solution in terms of the DSL's constructs, which now coincides with the vocabulary used by the domain's experts. Consequently the most difficult part in the GPL approach is now a straightforward mapping of your conceptual model into the DSLs.

It is important to remark that, for the incipient domain of automated learning scenarios, the prospect of *creating* new DSLs from scratch is just as much important as the possibility of choosing a best-match language among existing ones – as in step 2 above –, since it will enable learning designers to create new DSLs fitting their specific needs.

Even if the idea of having DSLs has been around for quite a while now and is more widespread than we might think – *lato sensu*, whenever someone uses an XML schema to drive his data or makes use of APIs in his program, he is effectively using a DSL -- these are quite "static" approaches, since the possibility of changing its conceptual model, while being

technically challenging, is something that cannot always be considered. Therefore, easily creating, modifying and incrementing, *à la volée,* abstract representations of systems and having it reflected on the final code is the challenge to be addressed by the LOP approach.

Certainly, the final code mentioned above has to be somehow generated – as opposed to hand coded. It is exactly the idea of having code generated, which resides one of the strongest appeals of LOP approaches: it means that DSL programming can be achieved by the domain experts, or lay programmers. The concept of *lay programming* concept is discussed in the next section.

4.5 DSL: enabling lay programming

With today's desktop programs easily reaching millions of source lines of code, software is collapsing under the weight of its own complexity. To cope with this complexity the software industry and academics have come out with methods, processes and even entire disciplines. However, these solutions are all based on the further honing of the programmers' skills, that is, they all require talented individuals who must craft ingenious ways of translating their original problems in computer terms, in an artisan-like way. In fact, it is not unheard-of to see in the literature authors comparing programmers with artisans:

> The software industry remains reliant on the craftsmanship of skilled individuals engaged in labor intensive manual tasks. (Greenfield & Short, 2004)

Acknowledging that such an approach is not scalable enough, proponents of the Language Oriented Programming envisage a future in which the software development process will be significantly simplified, to an extent that the domain logic of a system will be partly – ideally completely - developed by users. To this aim, DSLs are reserved an important role: just like high-level programming languages raise the level of abstraction in relation to assembly languages, DSLs raise the level of abstraction another step higher, becoming sufficiently close to the domain experts' reasoning and consequently enabling them, it is expected, to effectively program for that particular domain.

To (M. Fowler, 2005), "if such an approach", which he dubs **lay programming**, "could succeed, it would provide an enormous benefit", since this would come to removing the lack

of communication between domain experts and programmers, one of the “biggest roadblocks” in software development projects.

Presently, experts are important, but have no direct participation in the traditional software development processes, being their role to provide the information about their domain to developers in meetings held at the beginning of the process. Once developers are confident enough that they have “captured” all the knowledge of the domain, the domain experts are no longer required until about the end of the development, when they come back to certify that the application does conform to the initial requirements. Granted, this is a rough approximation of the process, as domain experts may intervene more often than that, most notably in the Agile development methods, like Extreme Programming (Beck & Andres, 2005), where a closer collaboration between experts and developers is advocated. But, as a rule, experts still play a rather passive role.

In the LOP approach, in turn, domain experts will be invited to play a much more consequential role, being directly involved in the development of the program. More than that, they will be given the right tools to make their job easier. Taking a look at Figure 22, we can better visualize these tools and how the LOP pradigm intend to go about the “lay programming”. As we will see in the section 4.7, there are several distinct implementations to LOP, but they all are, *grosso modo*, constituted by the blocks below:

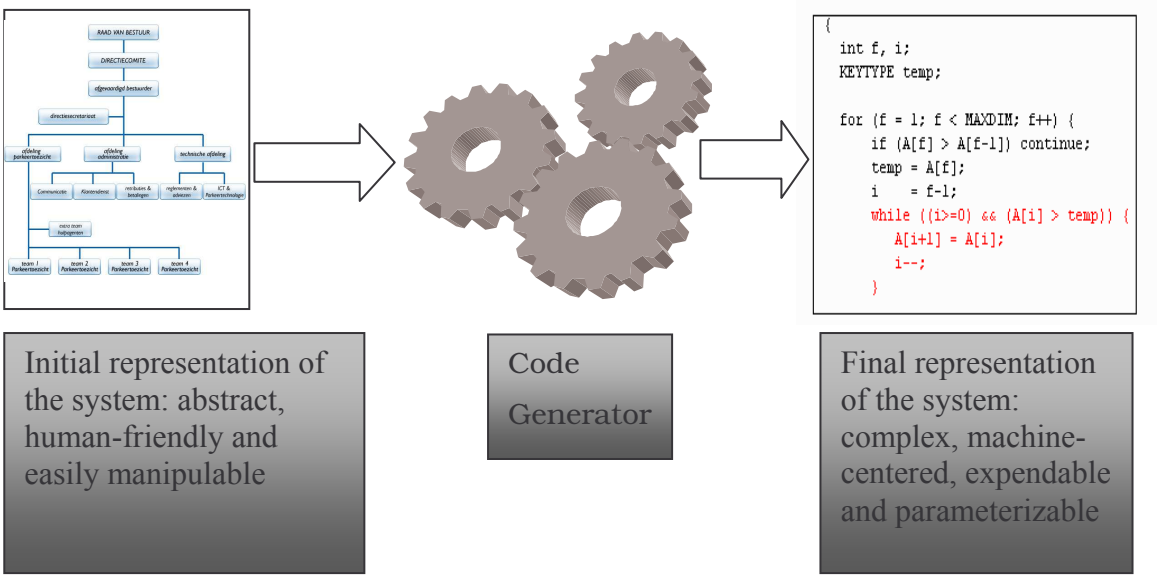


Figure 22: LOP programming

Placed at the left hand of Figure 22, domain experts will be offered an interface that is both user-friendly and conceptually expressive of their domain of expertise, making the *representation* of the system they want to conceive affordable to them. This initial representation, true only of the domain – i.e. completely unaware of the GPLs complex syntaxes -, is then fed to a program, called *code generator*.

At the right side of Figure 22 we have the programs, i.e. the system's complex representations intended to be executed by machines – and that nowadays are "crafted" by programmers. In the LOP vision, however, the elaboration of these programs will fall to other programs, i.e. the code generators, which will churn out the final application in an automated way. That is, this final and executable representation of the system is just a *function* – however convoluted the function is – of the initial representation. As a consequence, as long as we keep this initial representation, the program can be re-generated as many times as needed, being therefore expendable – it suffices to recreate it. And if the program does not meet the user requirements, it is the domain model that has to be reformulated.

The fact that programs are generated does not mean that we are dealing with an exclusively mechanical process though:

> [W]e propose a people-oriented approach to industrialization, one that uses vocabularies that are closer to the problem domain, and that leaves more of the mechanical and deterministic aspects to development tools (Greenfield & Short, 2004).

That is, instead of focusing on mechanical processes, LOP's "people-oriented approach" will further empower the role of the "domain expert".

Recasting this statement to the domain of learning design would mean to offer an *educator-oriented approach*, in which instructors and learning designers would have the entire control over the applications they use, since these would have been their own creations. That is, domain experts in education would be given a means to express their experience, their know-how, etc. in an automated fashion.

However, an approach that enables experts from a given field to easily make changes to their software without the constant aid of a programmer does not signify that programmers would be discarded of this process. In fact, according to LOP, programmers will be in charge of

writing the necessary supporting "meta" tools – e.g. the DSL editor and the code generator itself – that will allow domain experts to edit, generate and compile their "instance" programs.

We could also dispute here the affirmation that DSLs make development easier for domain experts not versed in language design, considering that language design "is a distinct (meta-) domain of expertise in itself" (Heering & Mernik, 2002), thus requiring specific competences. Certainly, for this task to be facilitated, "lay programmers" should be assisted by professional ones, what may give rise to a symbiotic association leading to a **participative design**.

Finally, it is believed that the direct participation of those who better know the domains to be represented will lead to more correct programs:

> The fact that users are able to code problems at the level of abstraction at which they think and the level at which they understand the specific application domain **results in** programs that are more likely to be correct, that are easier to write, understand and reason about, and easier to maintain (Gupta, 2002).

From what we have discussed, it is important to note that there is an "impedance mismatch" between real world concepts and computer constructs that has to be technically overcome, in a generative process, if we want to allow learning designers build programs. So, in the next section I will present some techniques used to solve this problem, i.e. how to allow the transformation of artifacts representing the conceptual model from any given domain into executable programs reflecting this model.

4.6 Generating software from DSLs

All language-oriented programming approaches implicitly or explicitly embrace the common ideal of relieving programmers from *"the highly mechanical and deterministic aspects of producing executables"*. Programs should be automatically churned out by development tools, so programmers can concentrate on the more "people-oriented" - hence nobler - art of creating and using problem domain vocabularies to solve their problems. So, it is expected that development tools adapted to the language-oriented programming paradigm be equipped with embedded code generation facilities that spare programmers from the *"rote and menial tasks"*.

If in the language-oriented programming, generating code involves a few automatic model-to-model transformations -- each new transformation giving rise to a new and less abstract model -- the last step is a model-to-code transformation. This last step requires specific technologies for code generation. According to (Dmitriev, 2004), the three main approaches to code generation are:

- *Iterative approach*: inspect each node in a source model, and based on that information, generate the target code.
- *Templates and macros*: define how to generate code in the target language.
- *Search patterns*: find where in the source model to apply transformations.

In order to show how the "magic" of passing from real world concepts to computer constructs can be done, I will provide a simple but illustrative example of the *iterative approach*. Suppose we want to write a simple program that tests for primes in a sequence of numbers that are kept in a text file. First, in a conventional example (Listing 2), the "model" (i.e. the list of numbers representing our "real world" concepts) is kept in a file (numbers.txt) and is "bound" to the program at execution time, when the numbers will be copied from the file and onto the data structure provided (a vector).

Now using the code generation iterative approach, the whole cycle is comprised of two moments: the *generation* time, corresponding to the execution of the code declared in Listing 3 and the *execution* time, corresponding to the execution of the code from Listing 4, which has been automatically generated by the code in Listing 3. At generation time, the source model will be inspected, i.e. the number.txt file will be read onto a vector and, based on this information, the generator - in this simple case, the sequel of instructions `System.out.println –` will create the correct target code (Listing 4). At execution time, the generated code is run, yielding the same result as in the original program (Listing 2).

Listing 2: A simple prime searching program

```
public class ListPrimes {

	static Vector<Integer> numbers = new Vector<Integer>();

	void getNumbers() {
		try {
			BufferedReader f = new BufferedReader(
					new FileReader("numbers.txt"));
			String numberStr = null;
			while ((numberStr = f.readLine())!=null) {
				numbers.addElement(Integer.parseInt(numberStr));
			}
		} catch (Exception e) {
			System.err.println("Unable to read from numbers.txt");
		}
	}
	public void listPrimes() {
		for (int j=0; j<numbers.size(); ++j) {
			boolean isPrime = true;
			int num = numbers.elementAt(j);
			for(int i=2; i <= num/2; i++) {			// see if num is evenly divisible
				if((num % i) == 0) {
					isPrime = false;		// num is evenly divisible -- not prime
				}
			}
			if (isPrime)
				System.out.println("number "+ num +" is prime") ;
			else
				System.out.println("number "+ num +" is NOT prime") ;
		}
	}
}
```

Listing 3: An iterative generator code

```
public class ListPrimes {

        static Vector<Integer> numbers = new Vector<Integer>();

        static void getNumbers() {
                try {
                        BufferedReader f = new BufferedReader(
                                        new FileReader("numbers.txt"));
                        String numberStr = null;
                        while ((numberStr = f.readLine())!=null) {
                                numbers.addElement(Integer.parseInt(numberStr));
                        }
                } catch (Exception e) {
                        System.err.println("Unable to read from numbers.txt");
                }
        }
        public static void main(String args[]){
              getNumbers();

               // Generate the program
               System.out.println("class ListPrimes {\n");
               System.out.print("    String numbers = {");
               for (int j=0; j<numbers.size(); ++j) {
                  System.out.print(numbers.elementAt(j)+",");
               }
               System.out.println("    };");
               System.out.println("\n");
               System.out.println("   public void listPrimes() {");
               System.out.println("     for (int j=0; j<numbers.size(); ++j) {");
               System.out.println("         boolean isPrime = true;");
               System.out.println("         int num = numbers.elementAt(j);");
               System.out.println("         // see if num is evenly divisible ");
               System.out.println("         for(int i=2; i <= num/2; i++) {");
               System.out.println("           if((num % i) == 0) { ");
               System.out.println("             isPrime = false;    // num is evenly divisible -- not prime ");
               System.out.println("           }");
               System.out.println("         }");
               System.out.println("         if (isPrime)");
               System.out.println("           System.out.println(\"number\" + num + \"is prime \"); ");
               System.out.println("         else");
               System.out.println("           System.out.println(\"number \"+ num +\" is NOT prime\") ;");
               System.out.println("     }");
               System.out.println("   }");
        }
}
```

Listing 4: the generated code

```
class ListPrimes {

   int numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   public void listPrimes() {
     for (int j=0; j<numbers.size(); ++j) {
          boolean isPrime = true;
          int num = numbers.elementAt(j);
          // see if num is evenly divisible
          for(int i=2; i <= num/2; i++) {
            if((num % i) == 0) {
              isPrime = false;    // num is evenly divisible -- not prime
            }
          }
          if (isPrime)
             System.out.println("number" + num + "is prime ");
          else
             System.out.println("number "+ num +" is NOT prime") ;
     }
   }
```

Certainly, much more processing can be added to the generator class than the simple calculations shown above – in this basic example, the generator took care of *file handling*, *iterating* over the file data and *buffering* of the "model". Thus, an extra advantage of code generation is that part of the processing will be transferred from runtime to generation time, consequently making the generated code more efficient.

If this simple example of the iterative approach offers a general view of what a generator does, in practice, scalable solutions are usually a combination of the other two approaches (*templates and macros* and *search patterns*), which results in a flexible solution using templates and complemented with efficient ways to match and select elements from the source model according to complex constraints. In Chapter Five, I will describe the MDEduc, a prototype that makes intensive use of template based programming.

In the next section, I will discuss the template engine technology and after this, I will present some of the most widely used implementations of the technology.

4.6.1 Templating Engines

Currently, template processing software is very popular in the context of development for the web. For example, the various server page technologies (Java Server Pages, Active Server

Pages, PHP, etc.) use templates to add dynamic content to web pages. However, another type of application that makes intense use of templating techniques -- and one that is particularly important for this thesis -- is code generation. In this case, templates are used to bridge the "abstraction gap", between specification and implementation:

> Template-based code generation is a very simple and powerful idea. Templates are intuitive because they are so close to what they generate and that is probably their main power. (Krzysztof Czarnecki, 2004).

Generating code through templates basically consists in producing source code output, e.g. Java source, which can then be compiled within a framework, for example. Template-based code generation is usually realized by *template engines*.

While programming with templates, programmers have to work with two "levels" of programming. The first one is the "meta" level, where the DSL, enriched with pattern-matching expressions and control logic, is used to access the different elements of the "data" – i.e. an instance model of the DSL - that will be used as part of the output. The second level, or the "program" level, consists of fixed code, referred to as "boilerplates", written in the target general purpose language, which will be merged in the output with the chosen data. This process is illustrated in Figure 23.

To facilitate the work with templates, a common way programmers make use of is to edit the output file in its final format -- i.e. in the target language, exactly like we would if we wrote the code manually -- and then to insert markers to indicate where we want the generator to insert the code that will be "executed" at generation time (as opposed to the actual execution time).

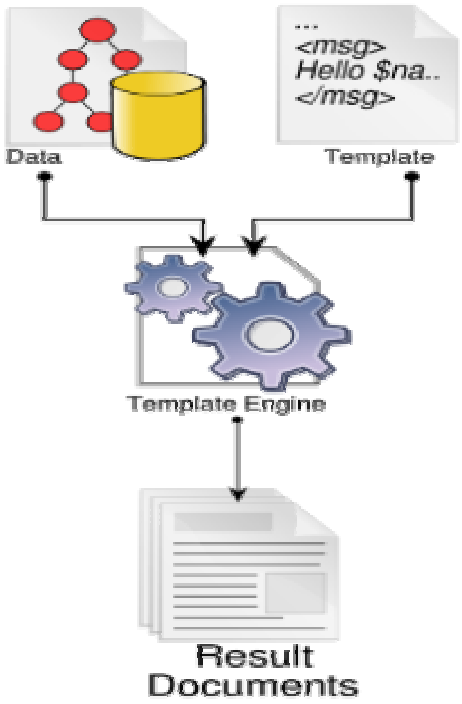


Figure 23: Template engine (figure licensed under Creative Commons)

4.6.2 Examples of template engines

There are many and varied implementations of template engines used to generate source code, of which I will show three examples: Velocity, XSLT and JET. Though this choice has been made based on my familiarity with this kind of tools, it is representative of the domain, since these tools significantly differ from one another, as we will see, thus covering a large part of the "template engine spectrum" as far as technical characteristics are concerned. For a comprehensive list of existing Java-based template engines, one may want to refer to this site[25].

4.6.2.1 Velocity

Velocity[26] is an Apache project offering a Java-based solution that consists of a template engine and a template language. It is mainly used for implementing the MVC design pattern in web applications and for generating Java source code. There is also a .Net version of it, the NVelocity.

---

[25] http://java-source.net/open-source/template-engines

[26] http://velocity.apache.org/

Compared to Java Server Pages (JSP)[27], Velocity presents a "cleaner" solution for web development, since it does not allow Java code to be embedded in pages. That is, contrary to JSPs, Velocity enforces a Model-View-Controller (MVC) pattern by separating Java code from HTML template code.

It has a built-in language, called Velocity Template Language (VTL), which is tailored to reference objects defined in Java code. That is, it permits to easily navigate through Java structures, so that, if you pass a list of Java objects you can, for example, invoke methods on these objects in a very convenient way. For example, if you had passed a Customer object as `$customer` to a Velocity template, you could access its properties and methods as follows:

```
$customer                    //reference to a variable
$customer.Name               //reference to a property
$customer.getAddress()       //reference to a method
```

As we can see, variables, methods and properties can be accessed through references - which will be replaced with their actual values at generation time. VTL is indeed a handy language that offers constructs like #set(), #if(), #elseif(), #else() and #foreach(), which can make the task significantly easier for developers. Also, VTL is *untyped*, which means that developers can write templates knowing nothing about Java types, getting along without type casting, etc. – though, knowledge about the structure of Java objects, as shown above, is still needed.

One of the inconveniences of working with template based programming, as already stated, is to have to work at both model and metamodel levels. In that case, it is highly recommended to use ad hoc template editors, if they exist at all. Fortunately, Velocity does have one, the Veloedit[28].

4.6.2.2 XSLT

XSLT[29] is a language for transforming XML documents into other XML documents, but it can also transform XML into source codes from different languages – though there are claims that "XSLT is too verbose to be an effective language for model-driven code generation" (van Emde Boas, 2004). While the Velocity engine has been conceived to accept java objects as

[27] http://java.sun.com/products/jsp/

[28] http://sourceforge.net/projects/veloedit/

[29] http://www.w3.org/TR/xslt

input, XSLT's was tailored to accept XML files. However, at their output, both can generate code alike.

In addition to the engine – already incorporated in many browsers – and the language, another key element of XSLT is the sub-language of patterns, the XPath, which is used for matching and selection purposes. An excerpt of an XSLT file, also called an XSL *stylesheet*, illustrating the XSLT's code generation abilities is shown below[30]:

Listing 5: Generating code with XSL stylesheets

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/XSL/Transform">
<xsl:output method="text"/>
<xsl:template match="/">
import java.awt.*;
import java.awt.event.*;

class <xsl:value-of select="//play/@name"/>Play extends Frame {
    /* The Props for <xsl:value-of select="//play/@name"/> ****/
<xsl:for-each select="//play/prop">
    Button <xsl:value-of select="@name"/>Prop
             = new Button("<xsl:value-of select="trait"/>");<xsl:text/>
</xsl:for-each>
…
</xsl:stylesheet>
```

As we can see, properties from the XSLT elements can accept XPath expressions (e.g. "//play/prop") making it easy to navigate an XML source file. XSLT engines can then combine a stylesheet, like the one above, with an XML file to generate a program, as illustrated in Figure 24 and exemplified in Listing 6.

[30] Example reproduced from the book "Program Generators with XML and Java" by J. Craig Cleaveland (Prentice-Hall, 2001)

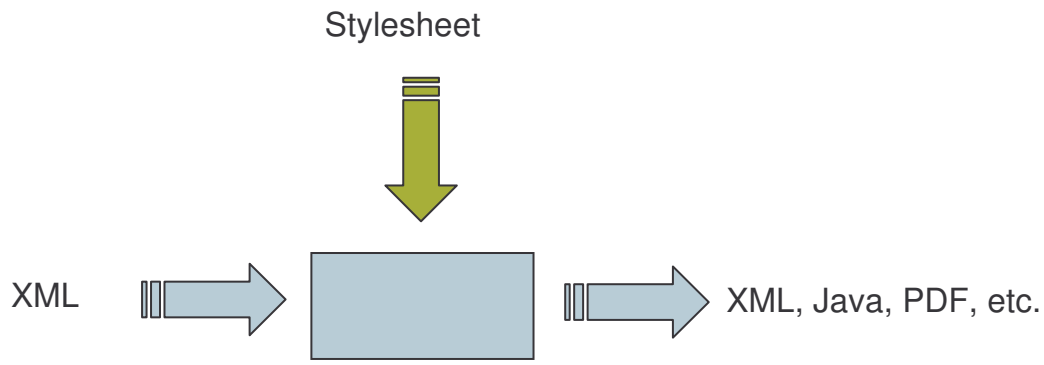


Figure 24: XSLT engine

```
<?xml version="1.0"?>
<play name="JackAndJill"
width="300" height="120"
start="bottom">
 <title>Jack and Jill</title>
 <prop name="up">
    <trait>Go up the hill</trait>
    <script goto="top"/>
 </prop>
 <prop name="fetch"/>
 <prop name="fall"/>
 <prop name="tumble"/>
 <scene name="bottom"/>
 <scene name="top">
  <addprop name="fetch">
   <trait>Fetch other Pail</trait>
  </addprop>
  <addprop name="fall"/>
  <addprop name="tumble"/>
 </scene>
</play
```

```
import java.awt.*;
import java.awt.event.*;

class JackAndJillPlay2 extends Frame {
    /* The Props for JackAndJill ****/

  Button upProp
   = new Button("Go up the hill");
  Button fetchProp
   =new Button("Fetch pail o'water");
  Button fallProp
   =new Button("Falldown,break crown");
  Button tumbleProp
   = new Button("Tumble down");


  …
```

Listing 6: XML source (left) and excerpt of the generated Java code (right)

4.6.2.3 Java Emitter Templates

Java Emitter Templates (Popma, 2004a, 2004b) is the Eclipse technology for code generation, widely used with the Eclipse Modeling Framework (EMF) for generating code driven by

models. It depends on the Eclipse Framework, what means that it cannot be used outside the Eclipse Platform. These two technologies are discussed at length in Appendix 1.

JET resembles Sun's Java Server Pages standard, implementing a subset of it. Its template files primarily contain fixed contents, or *boilerplates*, that are meant to be output exactly as it appears, mixed in with a number of JSP-like tags that are evaluated and interpreted by the template engine in various ways:

- Directives: They are, essentially, messages to the JET engine. There are two types:
  - *jet* directive: Declares the start of a template (syntax: <%@ jet attributes %>)
  - *include* directive: Includes another template (syntax: <%@ include file="URI" %>)
- Scripting elements, *scriptlets* and *expressions*:
  - Expressions: Substitute a Java expression result (syntax: <%= *Java expression* %>)
  - Scriptlets: Execute the code fragment (syntax: <% *Java statement* %>)

Like JSP – and unlike Velocity -, JET templates, by allowing Java statements in both boilerplate and tags, do not enforce a clean separation between code declaration and implementation. In this case, once again, a good editor can be very helpful, when editing JET templates. Luckily, JET also has an editor of its own[31], providing features like syntax coloring, error highlighting and code completion, which make the edition of JET templates far easier.

The process of transforming JET templates into source code involves two steps: *translation* and *generation*. In a first moment the source template is translated into a Java version of the template (not to be confused with the final Java code). In a second step, JET uses this Java template to generate the final source code. This process can be better understood with the aid of Figure 25.

[31] The JET editor, developed by Joël Cheuoua, (JET developers can never thank him enough) can be found at http://www.eclipse.org/emft/projects/jeteditor/?project=jeteditor#jeteditor.

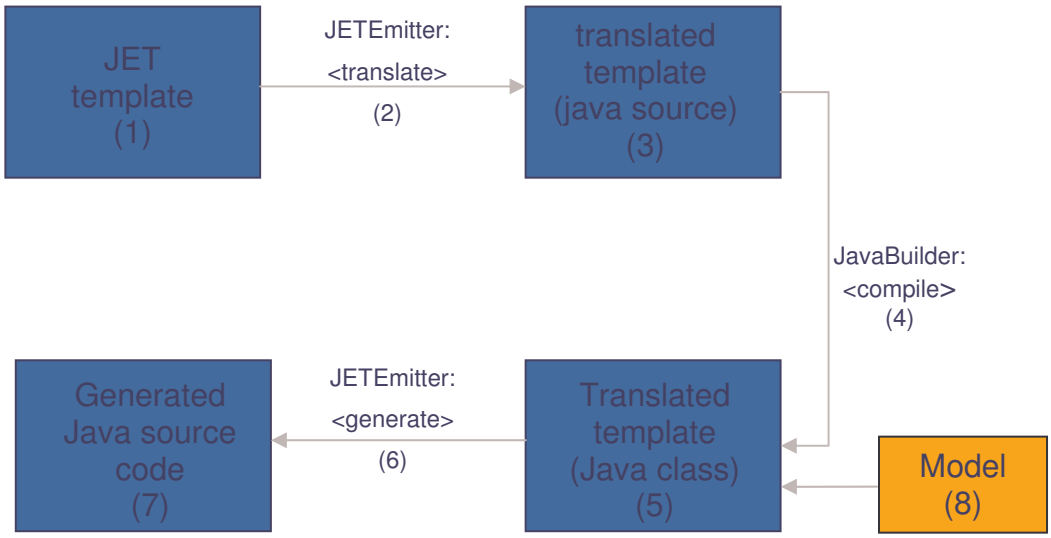


Figure 25: JET's translation and generation processes

Each step in this process is illustrated for a simple `helloworld` example (Listing 7). Although the JET API somewhat sophisticated, we can abstract much of its complex behavior using only the high level `JETEmitter` class:

> The `generate` method of this class translates a template to Java source code, compiles this source code to a template implementation class, asks the template class to generate text and finally returns the generated result (Popma, 2004b).

Suppose we create the simple `helloworld` template below [(1) in Figure 25]:

```
<%@ jet package="helloworld" class="TranslationDemo" %>

Hello, <%=argument%>!
```

Listing 7: `helloworld` JET template (1)

The engine translates each template into the source of a Java class that will mix fixed template data with the model-specific results of call backs. For example, the template above, once translated (2), would then yield the following source code (3):

```
package helloworld;

 public class TranslationDemo
 {
   protected final String NL =
System.getProperties().getProperty("line.separator")
;
   protected final String TEXT_1 = "Hello, ";
   protected final String TEXT_2 = "!";

   public String generate(Object argument)
   {
     StringBuffer stringBuffer = new StringBuffer();
     stringBuffer.append(TEXT_1);
     stringBuffer.append(argument);
     stringBuffer.append(TEXT_2);

     return stringBuffer.toString();
   }
 }
```

Listing 8: The helloworld template translated into Java (3)

The `JETEmitter` also calls (4) a java build (more specifically it creates a special Eclipse hidden project called `.JETEmitters` where it saves all java templates it creates and calls `build` on it), what gives rise to the compiled version of the java template (5). We refer to the latter as *static templates*, while we call the raw JSP-like form *dynamic templates*.

Note that, thus far, nothing about the model (8) that is supposed to "drive" the generation of the final code was mentioned. However, the template above provides a `generate` method with a single input parameter, conveniently called **argument**, which will be used to pass the model to the template. The Java template class is then "executed" (or "emitted", to use JET parlance) by calling its `generate` method (6), whose output will be a `String` containing the source code of the generated application (7).

Finally, there is an enhancement proposal for JET (referred to as JET2) that improves on the existing version in the following ways (Elder, 2005):

- Expand the JET language to support custom tags (which are distributed in "tag libraries). (The language specification will be described in a separate document.)
- Define Java interfaces and Eclipse Extension points for declaring custom tag libraries.

- Provide Standard JET tag libraries that make it possible to create entire transformations without recourse to Java and the Eclipse APIs. (These tag libraries will be described in a separate document.)
- Provide Eclipse API and UI for invoking such transformations.
- Provide a JET template editor

I have just described three templating technologies, but there is a large and fast increasing number of template languages – a Google search on *"template language"*, for example, yields hundreds of thousands of hits. But I had to make a choice for one of them for the implementation of the prototype described in Chapter 6 and in the end I opted for the JET solution. Since any specific option would have its pros and cons, I decided to privilege my experience in using these tools.

We arrive then at a turning point of this chapter. All we have seen up to now was a - rather extended - preamble aimed at clarifying some concepts required for a reasonable understanding of what *language oriented programming* is like and how it could help us with the two intents stated in the beginning of this chapter:

- Allows the easy creation of languages describing learning scenarios.
- Allows the easy creation of applications reflecting the aforementioned languages.

In the next section I will finally describe 3 different implementations of the LOP paradigm, Model Driven Architecture, Eclipse Modeling Framework and Software Factories, showing their idiosyncrasies and commenting on what each can contribute to the domain of learning design. Three other LOP implementations (Literate Programming, Intentional Programming and Generative Programming) can be seen in Appendix C.

4.7 Some proposals for implementing the LOP paradigm

All that has been exposed in this chapter up to now was aimed at explaining theories about domain specific languages (DSLs), its related concepts and some technologies that use DSLs to generate code, in a novel programming paradigm, here called Language Oriented Programming. This section will now list some of the existing implementations that use all

these theories to offer software developers – and domain experts – convenient ways of exploiting the LOP paradigm.

Trying to compile all or at least most implementations of the LOP is not a simple task, though. To the best of my knowledge, there is not a single work in the available literature where implementations of the LOP technique have been scrutinized and compared. In my research, all I could find was the original works describing each implementation separately and sometimes citing one or two other implementations, to give examples of related work. So, I decided to investigate the main technologies addressing the LOP tenets provided by at least the key players of the software industry and by some worldwide notable research projects. The technologies considered in my research were: OMG MDA (Frankel, 2003), Eclipse EMF (Budinsky, Steinberg, Merks, Ellersick, & Grose, 2003), Microsoft Software Factories (Greenfield & Short, 2004), Literate Programming (Knuth, 1992), Generative Programming (Krzysztof Czarnecki & Eisenecker, 2000) and Intentional Software (Simonyi, 1996).

However, along the investigation, I discovered that some of these technologies were in a more advanced stage of research and development (more mature, bigger user base, etc.), though none of them has thus far imposed a mainstream acceptance. So, to keep my research efforts to manageable proportions, I decided to focus on only two of these technologies, namely MDA and EMF. Reflecting this decision – and also to hold the size of this chapter to reasonable limits – I will only treat these two more deeply investigated technologies plus a third one – Microsoft's Software Factories - thus including a player that is always important in the market. For a glance on the other technologies, please refer to the Appendix C.

4.7.1 OMG MDA

The Object Management Group's proposal for the language oriented programming is called Model Driven Architecture, or simply MDA (Frankel, 2003; Kleppe et al., 2003). According to the official MDA Guide (MDA, 2003), MDA provides an approach and tools for:

- specifying a system independently of the platform that supports it,
- specifying platforms,
- choosing a particular platform for the system, and,

- transforming the system specification into one for a particular platform.

To specify systems and platforms, MDA uses models. That is, with MDA, models are more than just documents intended for human consumption, but artifacts that will be further processed by tools, giving rise to another model or to executable code:

> MDA is about using modeling languages as programming languages rather than merely as design languages (Frankel, 2003).

MDA proposes to classify models into different categories, according to viewpoints. First, at a very abstract level, there is the Computational Independent Model (CIM), which does not show details of the structure of systems, basically reflecting the vocabulary that is familiar to the practitioners of the domain in question:

> The requirements for the system are modeled in a computation independent model, CIM describing the situation in which the system will be used. Such a model is sometimes called a domain model or a business model. It may hide much or all information about the use of automated data processing systems. Typically such a model is independent of how the system is implemented. (MDA, 2003)

An example of a CIM for the domain of learning design could be the Peer Feedback Pedagogical Pattern, described in Section 2.3.1 or any textual notation from which a vocabulary of the learning scenario can be elicited.

Right below the CIM, considering the abstraction level, comes the Platform Independent Model (PIM). It exposes the domain-based structure of a system while hiding the details necessary for any specific platform. That is, a PIM "shows that part of the complete specification that does not change from one platform to another". The Peer Feedback DSL sketched in Figure 18 is a good example of a PIM.

Finally, at the bottom, there is the Platform Specific Model (PSM), which surfaces the details of the different platforms for which the system will be built. An example of a PSM could be obtained by adding platform specific information to our Peer Feedback DSL (e.g. specifying which files will be generated, where they will be saved, how the application will be deployed, etc.)

Another important concept in MDA is that of *transformation*. The idea is that domain-based abstract semantics, through successive transformations, give rise to applications. For example, the less abstract PSM is supposed to be obtained automatically, by transforming the corresponding PIM - in fact, the PSM is realized by transforming both the PIM and the platform model, as illustrated below:

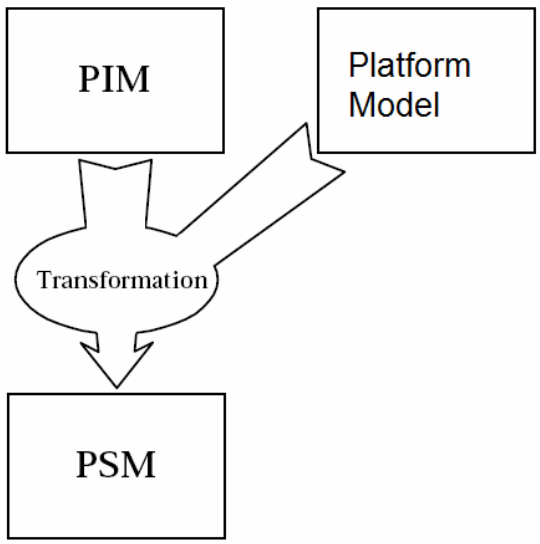


Figure 26: MDA model transformation (source : MDA , 2003 – adapted)

Except for the CIM, which is normally text based, all models in MDA should be expressed using either the Unified Modeling Language (UML) or a standard meta-metamodeling language called Meta Object Facility, or MOF (MOF, 2002). MOF is a language that, like UML, provides features for specifying the entities of a model; in fact, MOF is essentially "a subset of the class diagramming part of UML" (Martin Fowler, 2005).

MOF is typically designed as a four-layered architecture (Figure 27), though MOF's meta-levels are not fixed (i.e. the typical four meta-levels could be more or less than this, depending on how MOF is deployed). To avoid confusion, instead using the "meta-" prefix – like in "metamodel", "metaclass", and "meta-metaclass", etc. - MOF uses a phraseology like "an M1-level instance of an M2-level Class".

Thus, at the top layer, MOF provides a meta-metamodel, called the M3 layer. This M3-model is the language used to build metamodels, which MOF calls M2-models. Some examples of a Layer 2 MOF model are the UML metamodel (UML, 2004) and the IDL metamodel (IDL,

2002) or even a domain-based metamodel, like our Peer Feedback language. In turn, M2-models describe elements of the M1-layer, or M1-models. These would be, for example, models written in UML or a specific learning scenario created with the Peer Feedback language.

To use the terminology developed in Chapter Two, while M2-models are conceived by "upstream" designers, M1-models are crafted by "downstream" designers. If, for example, the upstream designers of a Peer Feedback language decide to create a rule stating that Receivers can pass along artifacts to Reviewers, downstream designers specify when and how this is done in a specific M1-model.

The last layer is the M0-layer or data layer, which represents the real world entities. In our Peer Feedback example, an M0 element would be actual people, actual artifacts, etc. (e.g. the Reviewer Student role is attributed to John, the artifact to be reviewed is the text file uploaded by Rachel, and so on).

Figure 28 illustrates the different MOF levels involved in the representation of a student record.

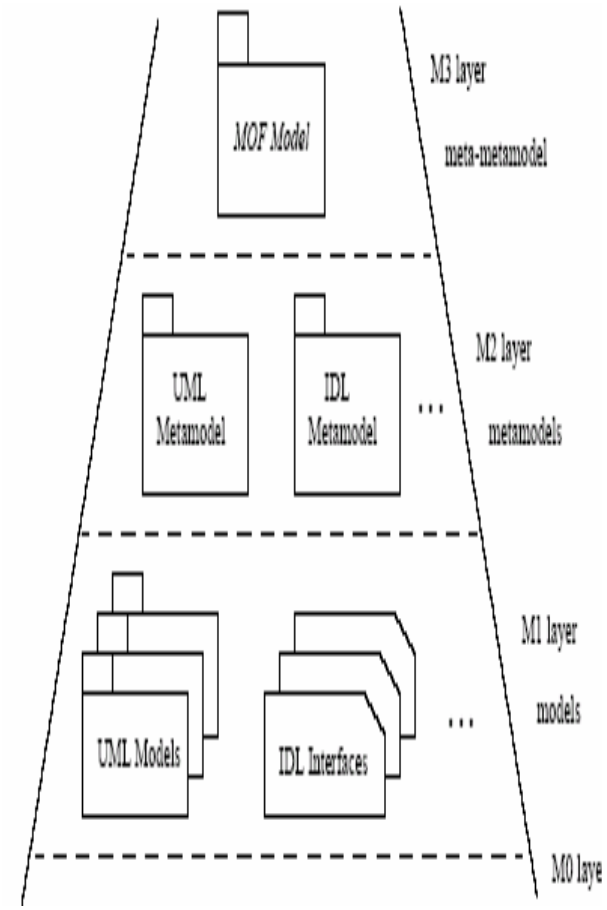


Figure 27: MOF metadata architecture

[Meta-metamodel (M3)]
MOF

[Metamodel (M2)]

```
<Class name="BusinessEntity">
  <Attributes>
    <Attribute name= "ID" type="GUID"/>
    <Attributename="type" type="String"/>
    <AttributeList name="Property">
        <Attribute name="name" type="String"/>
        <Attribute name="type" type="String"/>
    </AttributeList>
  </Attributes>
</Class>
```

[Model (M1)]

```
<BusinessEntity ID= "3564-AB54" type="Student"
    <Property name="name" type="String"/>
    <Property name="ID" type="IDType"/>
    <Property name="address" type="String"/>
    <Property name="dob" type="Date" />
</BusinessEntity>
```

[Information (M0)]

```
<Student name= "Mark" ID="8532"
            address= "ABC"
            dob="25-08-1970"/>
```

Figure 28: MOF layers (example)

In addition to UML and MOF, MDA relates to other standards, including the XML Metadata Interchange, or XMI (XMI, 2005) - used to generate XML schemas from MOF metamodels and to serialize models as an XML document conforming to a particular MOF metamodel – or the Query/Views/Transformations (QVT), a standard for model-to-model transformation (QVT, 2005).

Present Status and applications in the learning domain

Nowadays, MDA is one of the most widely disseminated implementation of the LOP ideas, presenting one of the most active communities, especially in the academic field. The list of MDA tools developed by vendors and open source projects is extensive and France has a particularly strong participation with contributions like Atlas' ATL project (ATL, 2007), Triskell's Kermeta metamodeling tool (Kermeta, 2007), LIFL's ModTransf model

transformation engine (Dumoulin, 2004) and LIP6's ModFact Toolbox (ModFact, 2003) to name but a few.

Considering the area of learning design, MDA is possibly the LOP implementation that has most contributed to the domain, though contributions come mostly from research projects still in very inchoate stages and, consequently, still not ready for prime time.

Incidentally, the **Laboratoire Trigone** was the first laboratory in France to have applied the MDA principles to this area, and more specifically to the conception of groupware systems, using the RAM3, a MOF-based graphical metamodeling tool (LePallec, 2002). While RAM3 - and now its successor, ModX - is a generic tool that can be used for any domain[32], in practice it has primarily been used by learning designers to create pedagogical metamodels and instance models of their scenarios.

In a more recent work, (LePallec, De Moura Filho, Marvie, Nebut, & Tarby, 2006) propose a generic support for guiding practitioners in the conception learning scenarios. Based on the assumption that the IMS-LD modeling activity does not count among the current practices of teachers, a formal framework (called Incremental Modeling Process) has been designed and implemented as a ModX module that will see teachers through the conception of learning scenarios.

In this same laboratory, (Caron, LePallec, & Sockeel, 2006) put forward the BRICOLES project, which proposes "a framework to deploy learning scenarios on e-learning platforms in a model driven approach", while (Renaux, Caron, & Le Pallec, 2005) propose a model driven approach to build components for Content Management Systems and Learning Management Systems. Other initiatives in France exist that advocate the use of MDA principles to the area of education. For example, (Thierry Nodenot, 2005) analyses the possibilities of applying the MDA approach (or, more specifically, the Model Driven Engineering - MDE) to the development of learning applications, with a special emphasis on the problem-based learning scenarios; (El-Kechaï & Choquet, 2005) propose a model driven analysis approach for the re-engineering of e-learning systems and (Laforcade, 2007) presents also related work.

[32] In fact, by definition, this observation could be applied to any LOP-based tool.

Outside France, other projects have also been identified that dwell in this same "crossroads" of learning systems and MDA, like, for example, (Dodero & Diez, 2006), (Grob, Bernsberg, & Dewanto, 2005) and (Wang & Zhang, 2003).

MDA's visibility makes of it one of the most criticized approaches. The focal point of its critics is that MDA lies on brittle foundation:

- UML, which is not formal enough (Martin Fowler, 2005; Greenfield & Short, 2004);
- MOF, which offers no concrete syntax allowing users to interact with (Cook, 2004);
- incomplete and unstable standards (Haywood, 2004; Thomas, 2003);

Still other quibbles can be found in (MDA_Wikipedia, 2007). These criticisms should be in part redeemed if we consider MDA's incipient stage:

> MDA is still in its infancy. […] A lot of the potential of MDA has still to be developed. The industry as a whole does not yet have enough experience with MDA to recognize the pitfalls. We have to be a bit patient. Object-orientation has taken about fifteen years from infancy to mainstream technology. MDA has the same or even greater potential of changing the way in which we develop software.

However it seems clear that whenever OMG does not provide clear and complete standards, adopters may fall prey to vendor "lock-in". UML CASE tools have already demonstrated that, left to vendors, incomplete standards pave the way to non-interoperable toolsets.

4.7.2 Eclipse EMF

Developers opposed to the fuzziness of OMG standards and to the panoply of different MDA – not necessarily interoperable - implementations may feel comfortable with the Eclipse Metamodeling Framework (Budinsky et al., 2003). By offering a well defined meta-metamodel, a steady API, a functional framework and a huge set of tools, all of them leveraged by the popular Eclipse Platform (E. Gamma & Beck, 2003), EMF has attracted many developers around the world to constitute one of the liveliest communities discussing and implementing the LOP principles. Consequently, if originally it was considered just another MDA implementation, today EMF has grown to acquire a status comparable to the one of the technology from which it stemmed.

At the beginning, EMF entices for its simplicity. Newcomers feel immediately attracted when they witness their sketchy models become full fledged applications – by default, editors for the models – in perhaps the most productive minutes of their lives. For example, the Peer Feedback DSL described in Section 4.2.3 could be effortlessly created by a learning designer with basic knowledge on metamodeling. Moreover, with two or three additional steps (no more difficult than pushing buttons or selecting items from drop down menus) an editor conformant to Peer Feedback DSL would be generated that could be used by learning designers or even by domain experts to create peer feedback scenarios.

Granted, in a second moment this initial enthusiasm may vanish, once newcomers realize that only superficial knowledge will not suffice to implement the desired tailored application – an indication that EMF, and LOP implementations at large, still have a long way ahead. However it is certain that this "customizability issue" tends to gradually disappear, as new templates and transformation tools will be developed, delivering, respectively, more specific final code and more powerful and friendlier ways of customizing generated code.

As a modeling framework, EMF permits to construct metamodels for any purpose (like, for example, describing pedagogical strategies) and offers code generation facilities, allowing to build applications based on the created metamodel. Furthermore, EMF also proposes a dynamic API and a reflective API that permit to work with models and instances without the need of generating code. The reader may refer to Appendix B for a more detailed discussion on how this can be done.

EMF consists of three fundamental pieces:

- The Ecore metamodel: a notation to express metamodels using Oriented Object constructs;
- A metamodel conversion framework: allows to import the metamodel expressed in other Object Oriented notations, namely, Java interfaces, XML schema and UML;
- A code generation framework: tools to generate code conformant to an Ecore metamodel: they include a GUI from which generation options can be specified, and generators can be invoked –like the JET generator, discussed in Section 4.6.2.3.

These three major blocks subsume many other important facilities, like a runtime support for the model change notification, a persistence API supporting out of the box XMI/XML serialization and de-serialization of instances of a model, an efficient reflective API enabling to create EMF objects dynamically and manipulate them generically, a set of generic command implementation classes enabling editors to be built that support fully automatic undo and redo, and much more.

Appendix A and Appendix B provide a lengthier explanation of the Eclipse Framework and of the Eclipse Modeling Framework, respectively.

Present Status and applications in the learning domain

The Eclipse EMF project is vast and encompasses several different projects, each one deserving to be considered a LOP implementation in its own right (e.g. Open ArchitectureWare[33], Sympedia GenFw Generator Framework[34], IBM Model Transformation Framework[35] and many others). The French academy and companies are also present with several projects that lie on top of the EMF: Topcased graphical editor[36], Acceleo code generator[37], Merlin Generator[38], etc. Independently of the implementation, an outstanding characteristic common to EMF's projects is that they are all much more than just intentions, position statements or standard propositions, offering "out-of-the-box" products that can be readily used to implement LOP applications.

Considering the learning design domain, no project using the EMF technology has been found in any of the EMF project repositories[39], which makes of the MDEduc prototype, introduced in this thesis, one of the pioneer EMF applications in this domain– and in the educational domain at large.

---

[33] http://www.openarchitectureware.org/

[34] http://genfw.berlios.de/

[35] http://www.alphaworks.ibm.com/tech/mtf

[36] http://www.topcased.org/

[37] http://www.acceleo.org/pages/accueil/en

[38] http://sourceforge.net/projects/merlingenerator/

[39]These repositories can be found at *http://wiki.eclipse.org/index.php/Modeling_Corner* and *http://www.eclipse.org/modeling/*.

Outside the traditional Eclipse-centered project repositories, other initiatives exist that exploit the potential of the tandem Eclipse/EMF to the area of learning design, though by staying out of these software "store windows" they lack more visibility. For example, (Vachet, 2006) investigate the dynamic adaptation of a learning scenario's services and content in relation to the context in which the scenario is executed. It seems, however, that this project is still in a very incipient stage since, according to the references found in their site[40], they limit themselves to the development of a metamodel, to the generation of the corresponding default EMF editor and to having OCL requests performed on the created models - consequently adding little to the benefits already provided by the original artifacts of EMF.

4.7.3 Microsoft Software Factories

A third implementation of the Language Oriented Programming paradigm comes from the Microsoft, with its Software Factories. As with any product coming from this giant software vendor, researchers cannot expect to have a leeway for investigation and much less contribution aimed at the development of this implementation, as the Microsoft's business model is to offer ready for use products.

According to (Greenfield & Short, 2004), Software Factories is a technology intended to deal with the two fundamental forces that make software development challenging: complexity and change. Microsoft envisions that these forces can be brought into control once the software development process will be dealt with, as in a factory assembly line:

> We can think of the software development process by analogy with the more general concept of a value chain, where each participant in the process takes inputs (which might be either physical goods or information assets) from one or more suppliers, uses their particular expertise to add value to these inputs, and passes on the outputs to participants further down the chain (Greenfield & Short, 2004).

Generally speaking, software factories represent a collection of guidance that helps architects and developers build specific kinds of application. For example, the Web Client Software Factory helps developers to quickly incorporate proven practices and patterns of building Web client applications. More formally, a Software Factory can be defined as:

---

[40] http://www.syscom.univ-savoie.fr/

> A **software factory** is a **software product line** that configures extensible tools, processes, and content using a software factory template based on a **software factory schema** to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components. (Greenfield & Short, ibidem)

Note that the word schema has a particular signification in this definition, meaning a description of the artifacts that have to be developed to produce a software product, in an analogy, for example, to an XML schema - which describes the elements and attributes that must be created to form a document. An example of a simple software schema is illustrated in Figure 29.

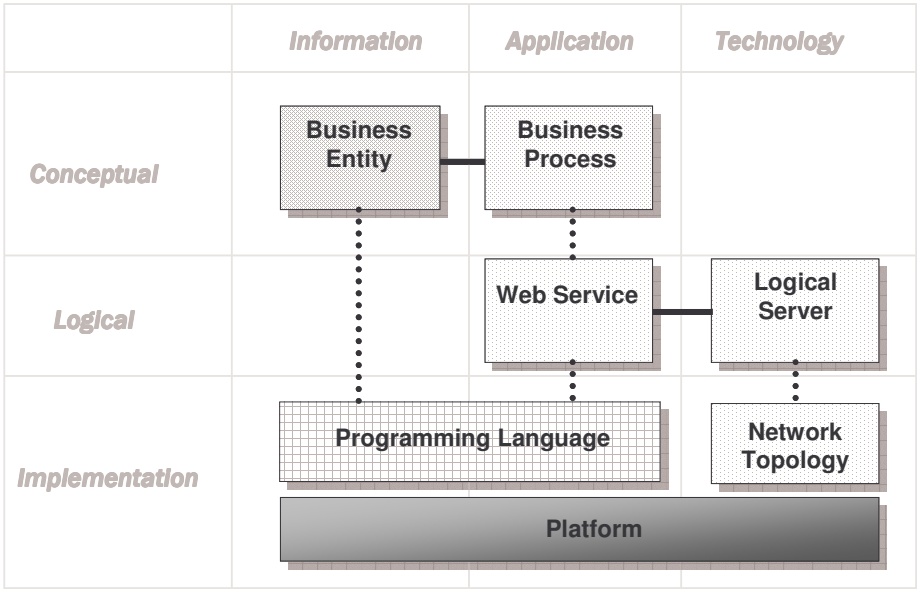


Figure 29: A Simple Software Schema (source: Greenfield & Short, 2003)

Software factories involve, in addition to language-based development, component-based development, software frameworks, design patterns, web services, etc. all of them brought together in an overall process drawing on the agile methods:

> There must not be any irreversible generation steps or large discontinuities between the model and the deployed code. It must be possible to change any visible artifact within the process and be able rapidly to recreate the end-result. (Cook, 2004)

In that case, Software Factories seem a more ambitious enterprise than other LOP approaches, since it proposes not only a model driven solution to software development, but the integration of an assortment of software design and development technologies, each one being

represented by a specific DSL in a modeling grid (i.e. the schema), in order to generate software families in specific domains.

Present Status and applications in the learning domain

Several implementations of software factories have been developed for different platforms. For example, the Microsoft Patterns and Practices Team (MPPT, 2007) has released four software factories for web clients, mobile clients, web services and smart clients whereas the Microsoft Services division has released the EFx Factory (EFX, 2007), which implements Service Oriented applications and services using the .NET Distributed Architecture. Other implementations by Microsoft and by other companies exist that generate applications for supplementary platforms. Each implementation is backed up by nascent communities, with forums, newsgroups, blogs, homepages, etc., constituting then an incipient user base.

No application in the domain of e-learning has been referred to in their homepages. However, the European Learning Grid Infrastructure (ELeGI) project proposes a Software Factory based approach (Gaeta, Ritrovato, & Gaeta, 2005) for developing e-learning applications for the grid computing technology (Foster & Kesselman, 1998).

4.8 Choosing a LOP implementation

What I needed for the Multi-EML was a technology that, even if still in a research stage, offered one off-the-shelf implementation, thus allowing me to put the language oriented programming principles into practice. Given the multiple possibilities, I could afford to take other aspects in consideration - like, for example, being free, open source, extensible, etc. - before choosing one of them. Table 2 summarizes all aspects investigated in the different implementations.

	EMF	Generative	Intentional	Literate	MDA	Software Factories
Free and open source implementations	Yes	Yes	No	Yes	Yes	No
Mature and proven technology	Yes	No	No	No	No	No
Community-backed projects	Yes	No	No	Yes	Yes	Yes
Support	Yes	No	No	Yes	Yes	Yes
Used in e-learning applications	No	No	No	No	Yes	No
Text Editor framework	Yes	No	No	No	No	Yes

Table 2: LOP implementations

Thus, considering all aspects above, I happened to find in the Eclipse Platform, and particularly in its modeling framework, the Eclipse Modeling Framework (EMF), an environment where those requirements could be addressed. Together, the Eclipse platform and EMF work in tandem so as to offer a cohesive structure: while Eclipse provides a powerful platform for integration at both the User Interface (UI) and file level, EMF enables applications to integrate at a conceptual level, thus allowing an even finer granularity.

Each of the items from Table 2 is discussed hereafter, emphasizing the EMF position in each:

Free and open source implementations

EMF is free and open source. Likewise, MDA, Generative Programming and Literate Programming offer at least one free and open source implementation.

Mature and proven technology

As a rule all LOP implementations are still in their infancy, even if some of them, like the case of Literate Programming, are available for a relatively long time. According to (Merks, Steinberg, Hussey, & Damus, 2007), EMF has been used since 2002, not only by individual users, but also by:

- Other Eclipse projects: Graphical Modeling Framework (GMF), EMF Ontology Definition Metamodel (EODM), UML2, Web Tools Platform (WTP), Test and

Performance Tools Platform (TPTP), Business Intelligence and Reporting Tools (BIRT), Data Tools Platform (DTP), Visual Editor (VE) and many others.

- Commercial offerings: IBM, Borland, Oracle, Omondo, Versata, MetaMatrix, Bosch, Ensemble, etc.
- Applied sciences: Darmstadt University of Technology, Mayo Clinic College of Medicine, European Space Agency, etc.

The fact that some of the software industry major players, like IBM, Oracle and Borland are investing on it and even redirecting the architecture of their products to take advantage of EMF's facilities has also contributed for my signaling it as the only mature and proven LOP implementation in Table 2.

Community-backed projects

With a large open source community and over 1,000,000 download requests in 2006, EMF is certainly the most active of the LOP implementations. Even if other LOP implementations are supported by lively communities, they are still far from the figures presented by EMF.

Support

EMF provides free and instant support. Contrary to other technologies, where free help, if any, is provided by other users, with EMF, "official" help is the norm. In practice, if you send a message to the EMF newsgroup, before any regular user makes a move intending to answer it, a detailed and precise response by one of the EMF developers (and particularly by Ed Merks) pops up in your newsbox. And they leave no message unanswered. Additionally, there are several books, articles, presentations, videos, etc. that offer valuable help to EMF adopters.

Used in e-learning applications

As already stated, I could not find in any of the traditional Eclipse-centered project repositories a single reference to applications or even intentions to use the EMF technology for e-learning applications and more particularly in the domain of learning design. The only one was an embryonic application that apparently does not go any further than the pieces of software generated by default by the EMF.

However, EMF presents a great potential to be used in the domain of education and it seems to be a matter of time before applications that will appear that permit to execute the countless models that learning designers will certainly come up with. In any case, the MDEduc prototype, described in Chapter Five is certainly in the avant-garde of the applications in the domain of education that make use of the EMF technology.

Text Editor framework

To the implementation of the MDEduc prototype, described in the next chapter, I also needed a text editor, adapted to the pedagogical patterns idiosyncrasies – or, at a bare minimum, to their syntax - where practitioners could document their knowledge. Thus, another important point that has been highly considered for the choice of the LOP implementation, was that the LOP development platform offered, firstly a framework that facilitates the development of text editors and then a way of integrating both editor and modeling modules without much effort. Only EMF and Software Factories, by making part of broader programming environments, qualified for this requirement.

Other more specific features offered by EMF can be listed. For example, EMF reuses the knowledge of UML, XML Schema and/or Java. That is, EMF does not force you to start afresh. You can enter the new LOP world with your old knowledge. In particular, you will be able to use all your modeling experience based on UML. We could well call EMF "UML programming" (even if this expression raises the ire of detractors of the idea of making UML a programming language[41] or even making it drive code). However, whereas UML is mainly used nowadays as a specification language for object modeling, EMF offers a stable, proven way of constructing metamodels using the UML class diagram's concepts and enabling these metamodels to be easily turned into full fledged applications just by "pushing a few buttons".

Furthermore, some EMF features (more thoroughly described in Appendix B) have been extensively used that facilitated the development of the prototype:

- Reflective API: analogous to Java reflection, but much more powerful, allows you to inspect your objects -i.e. obtain the object's metadata at runtime.

[41] For a discussion on how UML can - or cannot – be used to leverage the LOP approach, please see (Martin Fowler, 2005).

- Dynamic model definition: allows you to create your metamodel dynamically, without having to generate a single class.
- Persistence API: facilitates the round-trip tour between memory-based objects and file system resources.

Certainly, EMF is not a finished technology and presents some disadvantages. For example:

- At the moment, EMF implements only a subset of the MOF.
- By default, EMF only provides generations of Eclipse editors. If you want to generate other types of application – as it was the case of the MDEduc, which generates Web applications – you have to construct your own templates and program your own generator framework to deal with them. In other words, it is up to developers to extend the generator to capture their own patterns – which is not necessarily an easy task. However, other members of the EMF community, like the Acceleo Company[37], are well ahead the EMF development team, providing many different templates for varied technologies.
- Even if there are useful tools, like the Merlin Generator[38], which allow you to effortlessly map models to templates, thus facilitating the generation of code, the principle of lay programming, for the moment, is not recursive (i.e., it cannot be easily used to extend the platform itself, e.g. to develop the templates).

4.9 Conclusion

In Chapter Three, I proposed the Multi-EML, a new approach for designing learning scenarios that required the ability of the learning designers to create their own models and applications. As argued in Chapter Three, this approach improves on existing ones, namely EML and ISD, by offering a much more flexible method to conceive learning applications, but, on the other hand, it requires a new programming paradigm enabling domain experts to actively participate in the development efforts. This requirement was perfectly fulfilled by the Language Oriented Programming (LOP), which posits that the real opportunity for major strides in software development "lies in processing information captured by abstractions to automate development processes" (Greenfield & Short, 2004).

One of the missions attributed to LOP tools is to enable programmers to concentrate on the domain problem, transferring menial tasks to the machine. To achieve this, LOP's implementations provide two main features: firstly, they elevate the abstraction level of the programming languages, and secondly, they use programs that synthesize other programs, in a generative approach. By combining both techniques, LOP's implementations are expected to allow programmers to "build software of a complexity that cannot be approached using today's methods" (Tristram, 2003).

As we have seen, there are not many applications directed at educational domain. This suggests that the technology is not ripe yet. In fact a good indication of a technology's maturity is when it slows down the pace of investigations on core subjects and transfers some research thrust to "domain" problems. Even if LOP-based technology is still predominantly investing in the advancement of its nuts and bolts, it will certainly come through this stage soon - especially because it has been conceived with domain problems "in mind".

Chapter Five: The MDEduc Prototype

5.1 Introduction

In Chapter Two, I discussed the design of learning scenarios, showing a few notable ways commonly used by learning designers to create their learning models while in Chapter Three, I argued for the utility of having different learning models representing distinct aspects of learning scenarios. The question that followed was then how to implement an environment that enabled designers to work with multiple models – e.g. create, refine and transform them – while permitting to generate code out of them. So, in Chapter Four, I showed that the Language Oriented Programming (LOP) was a programming paradigm that presented exactly these preoccupations – i.e. enabling the creation of models and letting them drive the generation of the final application - as its main goals and, as such, was a perfect match for the mentioned requirements. Still in Chapter Four, I appointed the Eclipse Modeling Framework (EMF), among other offers coming from some of the key players in the software industry, as the optimal solution for my needs and justified this choice.

The present chapter will now present the **MDEduc**, a prototype built in the Eclipse Platform, and more precisely on top of the EMF, that offers an environment where the requirements listed above can be addressed for the domain of learning design. Thus, this chapter is the natural place to which the ideas discussed in the previous chapters will converge, hence molding a prototype that is supposed to reflect most of the propositions.

Essentially, MDEduc is an environment where education practitioners can describe their pedagogical strategies using any textual notation – although it offers extra facilities for creating strategies respecting the pedagogical pattern notation - and where learning designers may take over from them to create a formal model from scratch and to generate an application – as opposed to code it by hand. Also, with the aid of supporting tools, it is possible to perform transformations between any two models. In the educational domain, this means that a learning scenario expressed in the Peer Feedback metamodel, for example, could be

converted into, say, the IMS-LD metamodel (and consequently executed in the Coppercore runtime engine).

The remainder of this chapter is a description of the MDEduc at the level of its constituent parts, namely its four Eclipse plug-ins. However, in order to facilitate the reading of this chapter, making it more amenable to non-computer-savvy readers, instead of addressing each plug-in separately, mixing both high- and low-level details for each in a different section, I decided to break up the explanation in two parts. Firstly, sections 5.2, 5.3 and 5.4 will address high level questions considered in the design of the plug-ins - which means that they can be read by domain experts and learning designers - in other words, by the MDEduc's end user. Secondly, sections 5.5, 5.6 and 5.7 will provide an in-depth explanation of how these plug-ins have been implemented, and are hence aimed at a computer-proficient audience – for example, a programmer wishing to extend the prototype. Section 5.8 will discuss extra features provided by the MDEduc and Section 5.9 will describe a concrete example of the implementation of a learning scenario.

5.2 Different possibilities for modeling pedagogical scenarios

We saw in Chapter Two that there are many different forms of capturing, representing and automating the knowledge of experts in learning design, four of which I described more thoroughly: pedagogical patterns, models, design languages and scenarios. Nevertheless, they are not completely interchangeable, as each notation presents specific endowments that make it more or less suitable to each of the distinct phases in the learning design lifecycle.

For example, considering that pedagogical scenarios are conceived by people, it is reasonable to think that they will be first expressed in a natural language, in any textual notation. Thus, in such inchoate stages of the design, when practitioners -- not necessarily computer-literate -- are still conceptualizing their domains, notations like scenarios and patterns seem to be a better fit.

Having captured the knowledge in the form of a pattern, for example, how to go about making it run in a computer? Considering that our final goal is to have expressive automated scenarios, an ideal case would be to allow informal descriptions to "drive" executable programs - as pictured in Figure 30 (path 1) - that is, to automatically generate code that is

conformant to the informal description. However, at the present stage of development of the computer's science and technology we have nothing that comes anywhere close from this.

To make do with such a limitation, then, intermediate notations are provided that, while still true of the domain knowledge, are unambiguous enough so as to be used for automation purposes. Models and design languages are examples of such intermediate notations. Thus, as long as we cannot see a ground-breaking progress in the computer domain that allows us to take the shortcut 1, we are left with paths 2 and 3 to traverse. And to go this way, LOP seems to be our best bet.

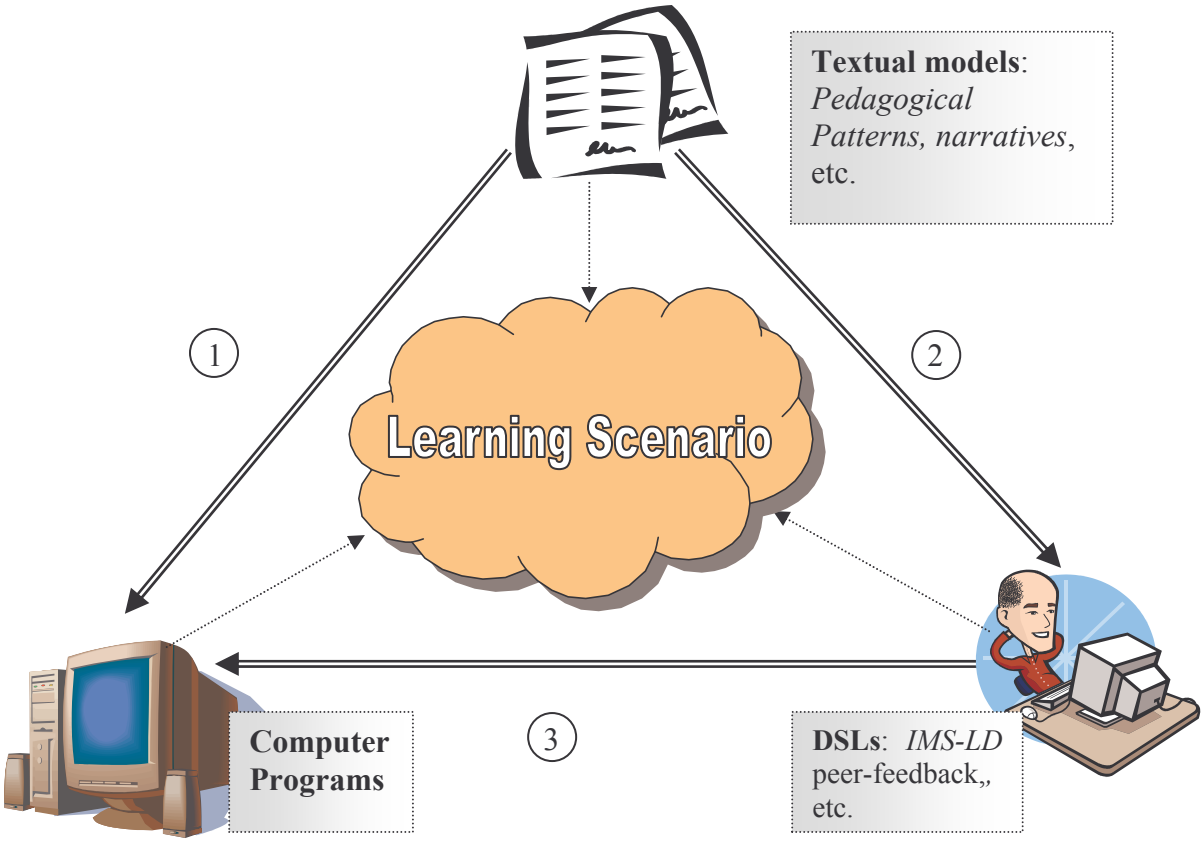


Figure 30: Different possibilities for modeling pedagogical scenarios

In the next section I will present the MDEduc, the prototype that will enable us to traverse paths 2 and 3 for the domain of learning design. It will be initially a bird's eye view, mainly emphasizing some high level considerations that guided the prototype's design, and then, in section 5.4, I will show some of MDEduc's typical use cases envisaged for the design of learning scenarios. The remaining sections will explain the low-level computational aspects of the prototype, tackling individually each of the four plug-ins that make up the prototype.

5.3 The MDEduc

Succinctly, **MDEduc** is a tool composed of four plug-ins built on top of the Eclipse Modeling Framework (EMF). It aims at facilitating the task of developing applications conformant to pedagogical strategies conceived by domain experts – e.g. by instructors. Essentially, considering all four plug-ins, MDEduc enables, first, to create informal representations of strategies using a notation based on the pedagogical patterns, then, to create a formal specification and, finally, to generate code of it.

However, while aiming at these three final objectives, MDEduc present some intermediary concerns that significantly influenced its design:

- Start the scenario design with an informal specification
- Allow for formal models to be derived from informal ones
- Keep formal and informal specifications "synchronized" and growing in parallel
- Allow for multiple perspectives
- Allow for formal models to guide code generation

The ensuing sections will discuss each of these concerns separately.

5.3.1 Start the scenario design with an informal specification

Many practitioners exist that are good at the domain they work but who are imbued with a misoneistic stance that prevents them from exploring facilities provided by new technologies - and computers in particular. This seems notably true of the domain of education. If we want to engage this kind of practitioner in the design efforts, we have to offer appropriate tools. In that case, informal representations – e.g. expressed in natural languages - are of paramount importance when it comes to capturing system requirements as they "support straightforward involvement and interaction of even inexperienced stakeholders" (Parets-Llorca & Grunbacher, 1999).

This requirement led me to envisage for the MDEduc a text editor, which practitioners may use to document their learning strategies. More than just a simple text editor, it is in fact a program that helps practitioners into editing *pedagogical patterns,* by providing ad hoc rules,

templates, etc. that guide those who are not accustomed to the syntax of this notation. Figure 31 shows a screen shot of the **Pedagogical Pattern Editor** (or simply **PP Editor**).

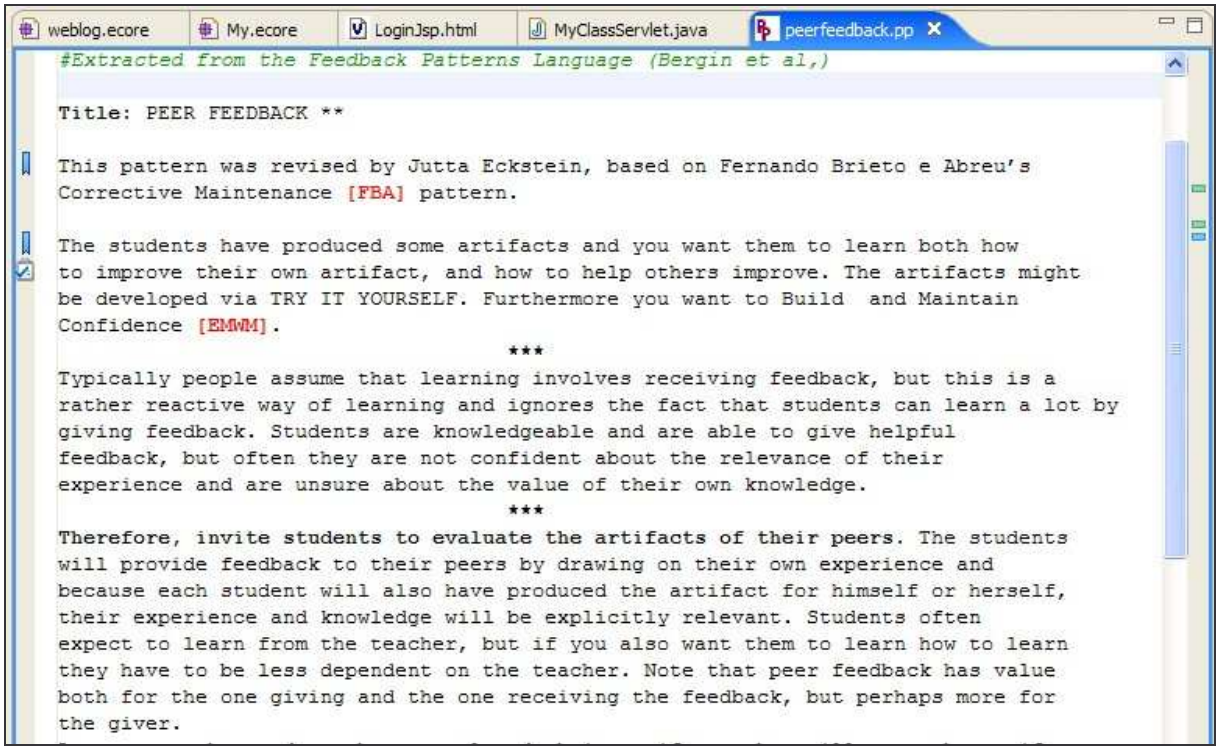


Figure 31: Pedagogical Pattern Editor (featuring the "Peer Feedback" pedagogical pattern - (Bergin et al., 2002))

As we can see, some parts of the above text are highlighted (in bold, in red, etc.). Behind each visual cue, there is a rule that formats the corresponding text, without any extra effort by the part of the person who is editing. For example, the Alexandrian pattern form requires that the first sentence of its *solution* section begin with a "Therefore" and be written in bold. So, the PP Editor can help the scenario writers in two ways: first, it offers a **template** for the solution section, second it provides a **rule** that formats the initial sentence appropriately (i.e. makes it bold).

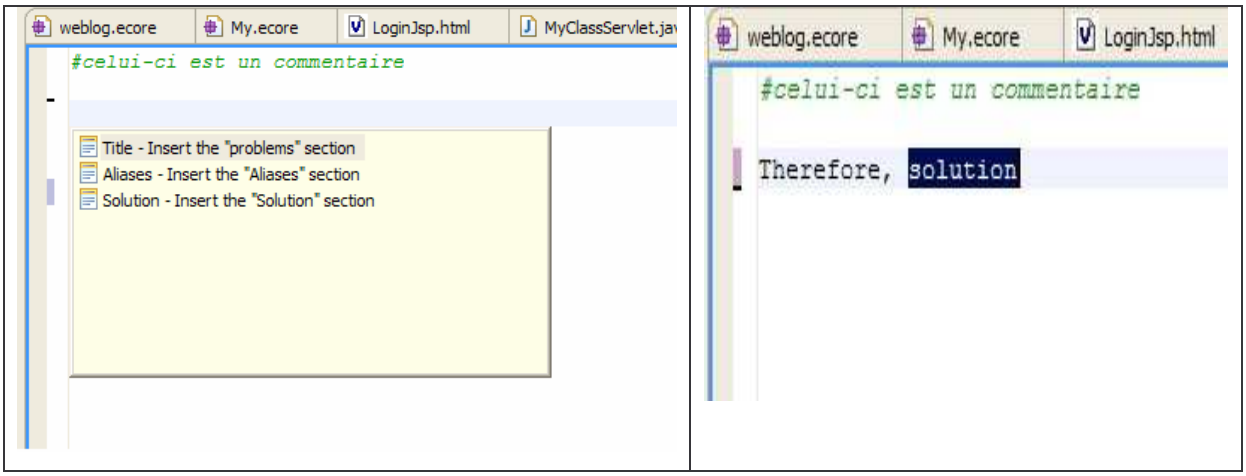


Figure 32: Pattern form section templates (left) and Solution template (right)

Thus, the user does not have to remember that there is a solution section and much less that it must begin with a "Therefore". All he or she has to do is call the solution template (Figure 32 – left), and it will automatically add the required "Therefore," and a box to be filled out (Figure 32 – right) with the actual solution. At the end of the editing, the solution rule will guarantee that the whole sentence be formatted in bold.

The PP Editor is composed of two Eclipse plug-ins: one is the editor itself and the other is the "model" plug-in, which keeps track of the elements of a pattern (e.g. title, aliases, links), consequently allowing these elements to be displayed in the Eclipse's **outline view –** which provides a glimpse on the structure of the pattern. More details on how this is achieved can be seen in Section 5.5 The Pedagogical Pattern Editor and Model).

5.3.2 Allow for formal models to be derived from informal ones

If informal notations help bring domain practitioners in to design efforts, they produce ambiguous and incomplete specifications that call for a complementary representation that lend itself to automation. Thus, it was necessary to envisage an editor for (semi-)formal specifications (e.g. models, DSLs, etc.).

However, instead of just providing an editor that let us handle concepts graphically – e.g. using 'boxes and arrows' or 'trees' – I decided to provide a tool that let us capture formal

specifications directly from the informal descriptions. This decision gave rise to the **EduModel** plug-in, which can be visualized in Figure 33:

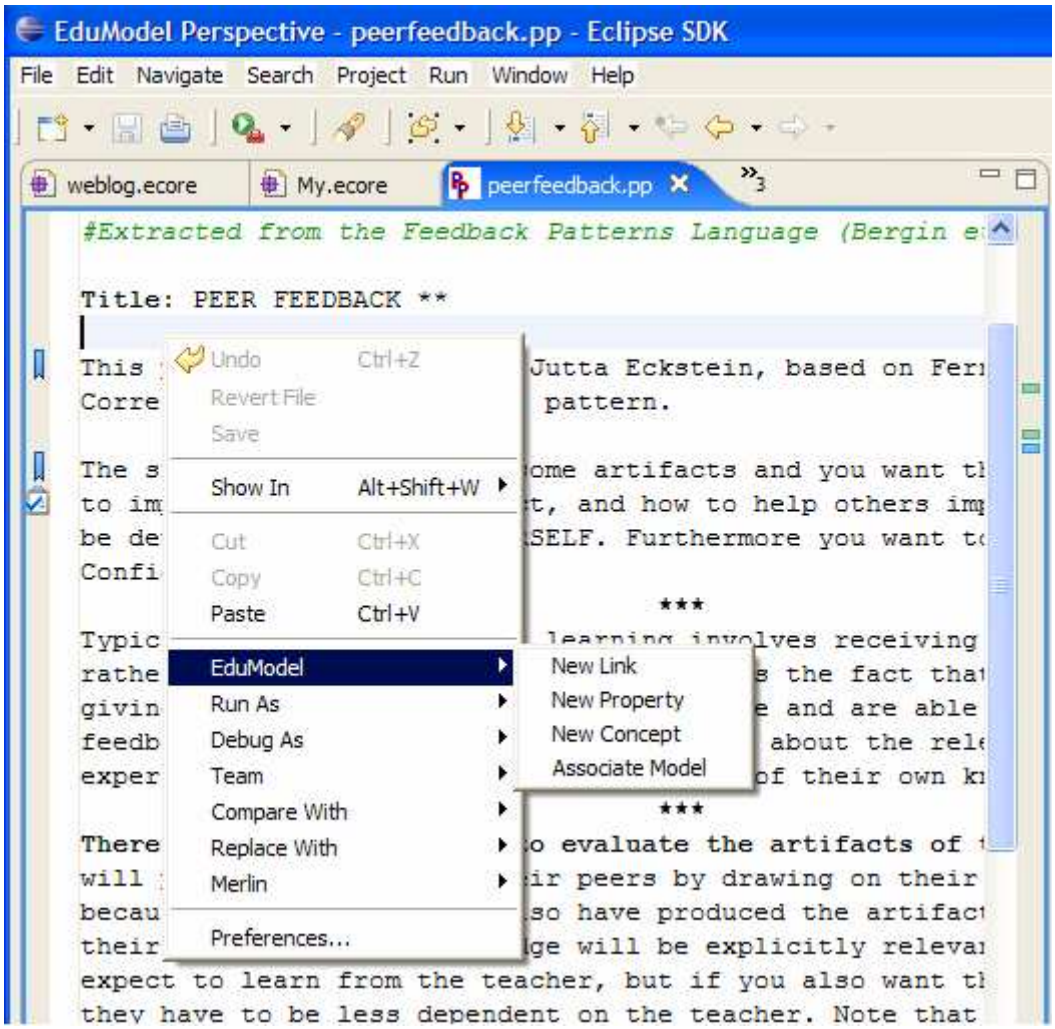


Figure 33: EduModel Actions

Strictly speaking, the EduModel plug-in does not have a user interface of its own, since it "borrows" the UI from the informal notation editors and views. Thus, what we see in Figure 33 is the EduModel's "actions", used to create the models, on top of the PP Editor user interface.

The advantage of this approach is that we can use informal descriptions from different sources, i.e. the same *actions* can be used with other editors or views. For example, the MDEduc also offers a special browser that allows models to be created (i.e. its concepts, links, attributes, etc.) directly from informal descriptions found in the Internet. This browser will be described in Section 5.8.1.

However, it is with the PP Editor that the EduModel works at it fullest, since, once created, the elements of the formal model will be displayed differently in the PP editor, meaning that the colored terms are elements that belong to a formal model (Concepts in red, Links in green, Attributes in blue). This can be seen in Figure 34.

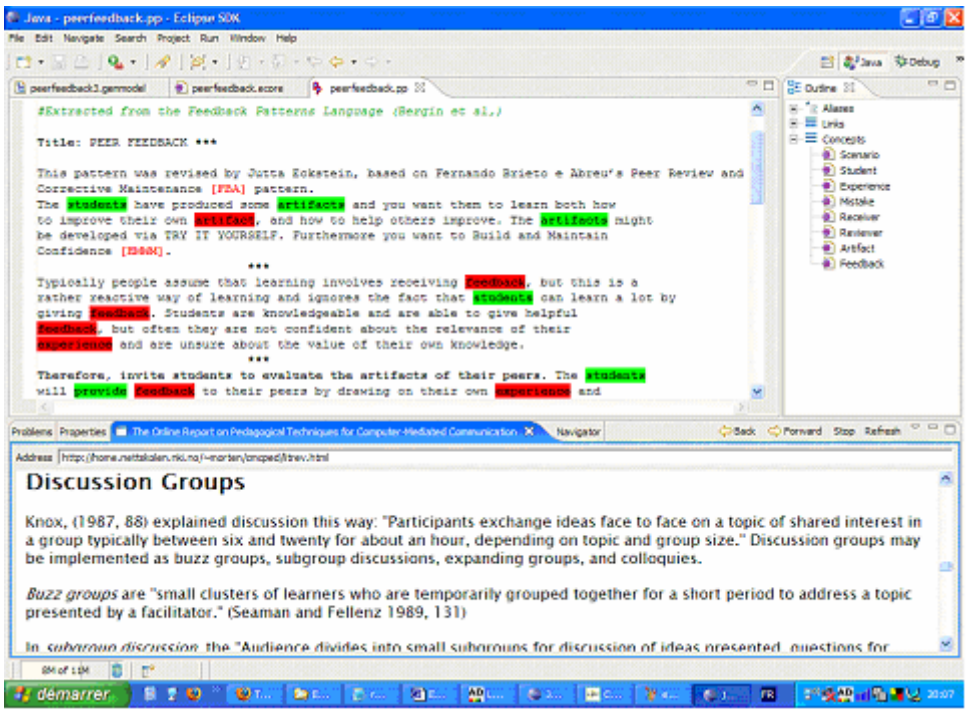


Figure 34: PP Editor showing elements from the formal model highlighted

After some iterations (creating concepts, attributes, etc.) a formal model reflecting the Peer Feedback metamodel is created. Figure 34 (right side) shows such a metamodel in the Outline View. In fact the metamodel is kept in an XMI file, saved under the name chosen by the designer. Figure 35 shows a small excerpt of the Peer Feedback XMI file:

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="peerfeedback"
   nsURI="http:///org.mdeduc.peerfeedback.ecore" nsPrefix="peerfeedback">
 <eClassifiers xsi:type="ecore:EClass" name="Scenario">
   <eStructuralFeatures xsi:type="ecore:EReference" name="students" upperBound="-1"
      eType="#//Student" containment="true"/>
   <eStructuralFeatures xsi:type="ecore:EReference" name="artifacts" upperBound="-1"
      eType="#//Artifact" containment="true"/>
   <eStructuralFeatures xsi:type="ecore:EReference" name="feedbacks" upperBound="-1"
      eType="#//Feedback" defaultValueLiteral="" containment="true"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="type"
eType="#//ScenarioType"/>
 </eClassifiers>
```

Figure 35: Peer Feedback metamodel (XMI file)

Alternatively, an existing model – in the Ecore format – can be associated with the informal description being edited, what causes all the model's elements to be colored in the PP Editor accordingly. These and other "use cases" will be further explained in Section 5.4.

5.3.3 Keep formal and informal specifications "synchronized" and growing in parallel

The process of transforming informal representations into formal ones is never perfect, always incurring in loss. It involves necessarily two critical moments: first the designer develops a particular interpretation on the intent of the creator, and then "freezes" this interpretation into a model, using a formal language. The problem is that this "interpret-and-freeze" process reflects circumstantial decisions that, even if appropriate for the moment they were taken, they may not endure afterthought considerations.

Considering that the primary and hence most truthful source of knowledge lies in the informal descriptions made by the domain experts – who are, after all, those who understand their domains -, once a system or even the intermediary model from which it originated does not convey the intention of domain experts, designers should "roll back" to the informal

descriptions to verify what went wrong. However this is not always possible, since the initial (informal) descriptions are not usually conserved:

> The problem with current software engineering is that we usually end up with a concrete software system, but don't know how we got there. Most of the design knowledge is lost, and this makes software maintenance and evolution very difficult and costly to perform (Krzysztof Czarnecki & Eisenecker, 2000).

Thus, I suggest that we should keep representations of the "phenomenon" in an informal notation, so as to leave a leeway for interpretation and negotiation before "freezing" it in a formal notation. That's why the MDEduc, by enabling the EduModel's actions only on top of text editors, indirectly "obliges" the safekeeping of the original textual descriptions (since they will be necessary to perform changes in the formal models). Having the original, relatively unconstrained view of the "system", we can always revisit later to check the consistency of derived metamodels or even, if none of the existing metamodels satisfy our needs, to create new ones.

Thus, another design decision implemented in the MDEduc is to force the conservation of informal notations even after formal specifications have been developed. This would afford two things:

- To keep a trace of the development of the system.
- To allow revisiting the original source in case derived artifacts (e.g. metamodels and programs) do not correspond to the envisaged needs.

Furthermore, if the idea is to capture the knowledge of domain experts through languages, patterns, narratives, etc. in both formal and informal notations, it is important to keep them synchronized so they, as much as possible, "tell the same thing" about the scenario they represent. This should not be confused with the capacity of representing different perspectives of the same scenario, when, evidently, different models will deliberately "tell different things" about the scenario, as explained in Chapter Three. Section 5.3.4, will discuss the question of modeling from different perspectives or paradigms.

MDEduc uses the notion of *project* to keep together related modeling assets, i.e. informal descriptions, formal specifications, generated code, etc. (for example, the Peer Feedback pattern text file, the Peer Feedback metamodel XMI file and generated sources). Furthermore,

it proposes mechanisms that help designers keep them synchronized, so that, whenever one given specification "grows", changes could also ripple into the other representations. Three mechanisms are involved:

- Synchronize between informal and formal models: observer deign pattern (via *extension points*).
- Synchronize between two different formal models that keep some semantic equivalence: for this, we need a supporting tool, like the Merlin Generator[38].
- Synchronize between formal models and code: the stock EMF code generation framework.

Technical details about how each of this performs its task will be given in the remainder of this chapter.

5.3.4 Allow for multiple perspectives

As discussed in Chapter Three, in such a complex domain like education, it is impossible to create a single model that captures all possible intentions of real learning scenarios. So, to make representations as expressive as possible, it is necessary to allow for multiple specific interpretations (here, in the form of models). If we intend to capture best practices of a given domain in formal models, we should enable the co-existence of multiple languages, because, otherwise, i.e. fixing a single language, would signify, at best, to capture the knowledge of the language itself and not of the domain.

MDEduc responds to this requirement with one of its primary functionalities, which is to permit the creation of models on the fly, whether referring to a single informal description or to several ones. Thus, different models may co-exist in the same project that reflect different perspectives of a learning scenario. Furthermore, with the support of the Merlin Generator, mentioned in the preceding section, different models can be related, or "synchronized" by creating transformations that capture semantic similarities between models, for example, which would be equivalent to "add" different and complementary perspectives of a given scenario.

5.3.5 Allow for formal models to guide code generation

This requirement is also handled by a primary functionality of the MDEduc, needing no external tool. Once the domain knowledge has been captured in any appropriate model, the next step is to generate – as opposed to hand craft – product families for the chosen domain in which the model can be executed, edited, etc. For this, MDEduc provides a tool that transforms models into applications, the **EduGen** plug-in.

EduGen allows mapping Ecore models to sets of JET templates corresponding to different applications, like editors, web applications, etc. If the templates for the desired application exist, an executable application is only a few mouse clicks away from a given model.

The different MDEduc plug-ins can then be mapped to the diagram of the Figure 30, originating Figure 36.

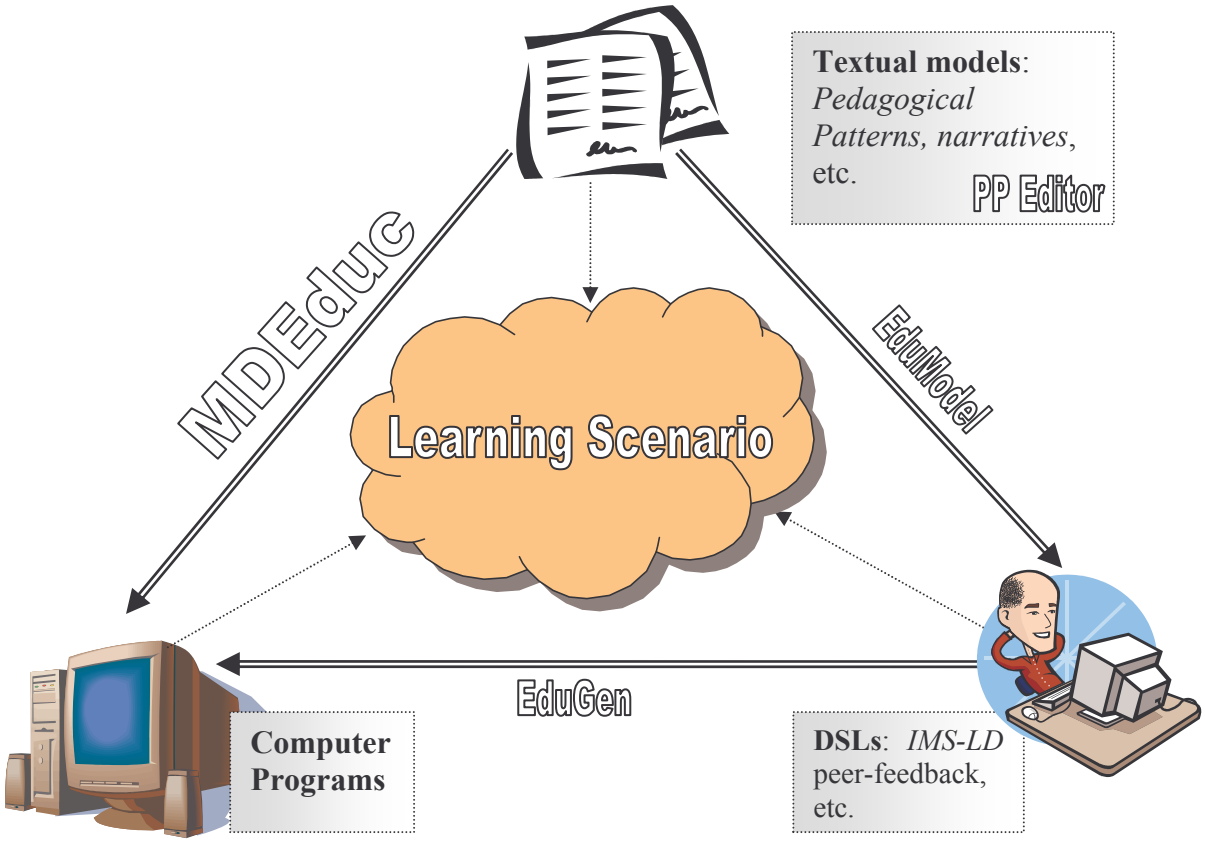


Figure 36: MDEduc in the learning design process

By permitting models to be easily created, preferably guided by informal descriptions laid out by domain experts, MDEduc offers an environment where the principles of lay programming

can be put into practice. For example, someone wishing to investigate on emerging theories and technologies that may contribute to improve teaching and learning, like, for example, *collaborative spaces* or the *future classroom*, might be supported by the MDEduc. Suffice it to say that at the end of the design process there will be a running application reflecting their ideas.

As a final remark, I would like to add that, even if MDEduc is a tool specifically designed with pedagogical scenarios in mind, it could well be used for other more generic modeling purposes. As already stated, this observation seems to be a recurring one among all applications drawing on the LOP ideas.

5.4 MDEduc typical use cases

5.4.1 MDEduc actors

An environment as MDEduc can support the design of learning scenarios in different ways and, to help visualize what and how different tasks can be performed, this section enumerates a few notable use cases. These use cases assume that there are distinct roles and actors involved in the development process and reflect the different competences that a user wishing to create automated learning scenarios should have.

The first role is the **scenario author,** who is usually an expert and a practitioner that interprets educational phenomena and describes them using usually a textual notation. To use the jargon of Chapter Two, the scenario author creates a specific design language. He or she may or may not be technology literate.

The second role is that of a **learning designer,** whose duty is to create and to enact a learning design model environment. Though, *lato sensu*, all roles involved in these use cases are *learning designers*, this one will receive this tag. His/her tasks include:

- designing the learning model – i.e. extracting from the informal description handed out by the scenario author a formal description that can be used for computational purposes;
- deciding the platform in which the scenario will be executed and generating the application for this platform;

- capturing the relations between different learning models in order to manage the transformation, integration, interchange and enrichment of learning design instances (i.e. data) described in different designs and scenarios. This activity requires that the learning designer the semantics of both target and source models.

The last role is the **programmer**, which, in practice, can be split into different sub-roles, including (but not limited to):

- Technology specialist/integrator - in charge of defining templates for code generation.
- Tester - in charge of testing the environment.
- Toolsmith – provides auxiliary tools, like for example, a fully functional application for managing, executing and editing the learning models.

It is important to note that, while the role of the programmer is instrumental, new scenarios can be created without the programmer intervention. Once the necessary templates for the required applications are available, scenario authors and learning designers become self-sufficient. Moreover, what sets learning designers apart from scenario authors is a design competence (more specifically the competence of eliciting a formal model out of a textual description). Considering that such competence can be easily developed, it is expected that instructors can play both scenario author and learning designer roles.

5.4.2 MDeduc use cases

Now that I have introduced the roles, I can delve into the main use cases envisaged for the MDEduc (please see Figure 37):

1. Editing an informal representation of the learning scenario;
2. Creating a learning design model from a textual representation;
3. Generating appropriate environment for a given model (editors, wizards, web applications, etc.)
4. Creating model transformations that captures the relationship between the different models;
5. Creating templates for existing software platforms.

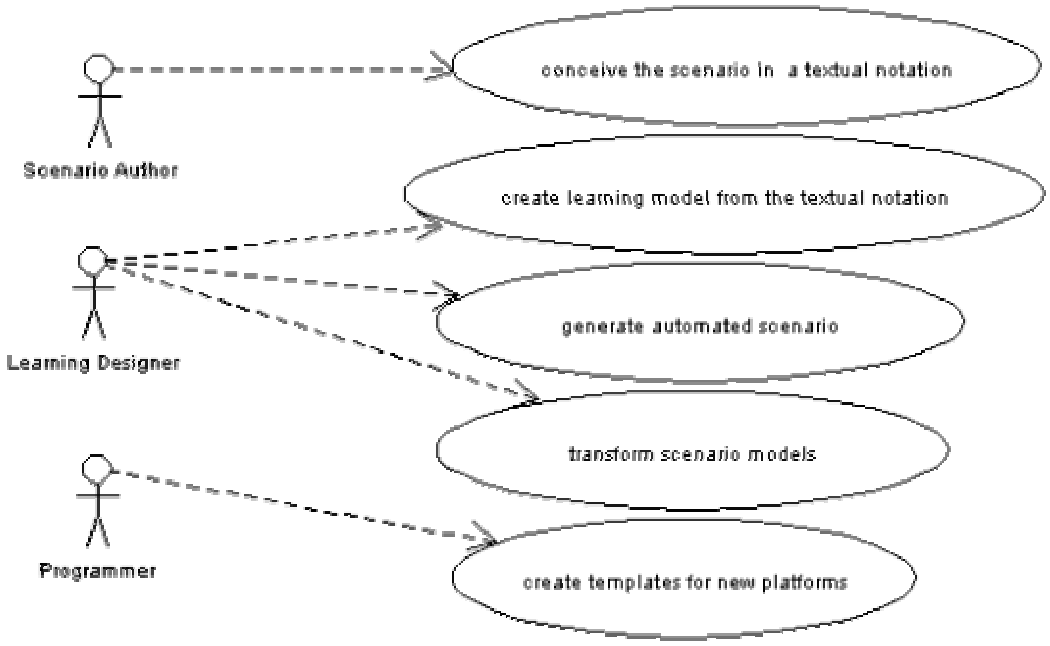


Figure 37: Simplified diagram with *use cases* and *roles*

5.4.2.1 Editing an informal representation of the learning scenario

In this step, practitioners experienced in developing teaching strategies – but not necessarily knowledgeable about computing – are in charge of writing a particular scenario down in any textual notation they find adequate. To this purpose, they can be supported by the PP Editor, which, as seen before, is a text editor that facilitates editions of scenarios respecting the pedagogical pattern syntax. It simplifies the task of the user by proposing content assistants, syntax coloring, templates, among other features, so that those who are not accustomed to the syntax of pedagogical patterns can also contribute without much effort. As we could see in Figure 31, some words and sentences are rendered differently, basically following – though somewhat arbitrarily adapted – the syntactical conventions common to pedagogical patterns. An example for the Peer Feedback pedagogical pattern has been shown in Section 5.3.1.

5.4.2.2 Creating a learning design model from a textual representation

In this step, learning designers are allowed to create learning design models from a textual representation. For this, they can use the PP Editor or the browser view specially designed to help the creation of EMF based metamodels. Figure 33 showed the Eclipse *actions* used to create the metamodel while Figure 34, in the outline view, showed the resulting metamodel

after a few interactions creating new concepts, their attributes and references and the same terms highlighted in the text editor (concepts in red, references in green and attributes –not shown – in blue).

While the model itself and its views (editor and outline) belong to different plug-ins, they are kept synchronized via the extension point/extension mechanism provided by Eclipse (i.e. all plug-ins declaring themselves as "listeners" –like the one to which the pedagogic pattern editor belongs - will be notified whenever there is a change in the model). This mechanism will be discussed in details in the next section. This specific use case has already been illustrated, for the Peer Feedback metamodel, in Section 5.3.2.

5.4.2.3 Generating appropriate application out of a given model (editors, web applications, etc.)

In this use case, learning designers will generate a full fledged application from the pedagogical formal specifications previously created. For this purpose, MDEduc comes provides templates for a web application, as we will see in action in Section 5.9. Also, using templates provided by the EMF framework, an editor for the created learning model can be easily generated. Any other software platform will require the implementation of its own templates, an activity proposed as future work in the conclusion of this thesis.

This particular use case will be illustrated with a complete example in Section 5.9, when a web application will be generated for the "What is Greatness" learning scenario (Dalziel, 2003).

5.4.2.4 Creating model transformations that capture the relationship between the different models

Suppose that we a have a Peer Feedback model or a CPM model (T. Nodenot & Laforcade, 2006) and we want to deploy them in the Coppercore platform, which understands only IMS-LD model. This is a typical situation in which users want to tackle the strategic and economical issues of interoperability between different design instances.

Whenever desirable, the learning designer can define transformations between elements from different learning design models that present a semantic relationship between them. This is an

added facility that stems from the fact that all created learning models (metamodels) are Ecore-based, which means that we can use some of the state-of-the-art model transformation technologies and tools to conveniently perform this task. For example, we can use the Merlin Generator, which provides an easy to use editor for creating mappings between the model elements (classes, attributes, etc.) and for creating mapping rules to control the transformations with a java-like scripting language. Additionally, models expressed in the UML notation (like the CPM, for example) may also benefit from this feature. A simple example of metamodel transformation (between a simple model built for the "What is Greatness" scenario and the IMS-LD specification) is provided in Section 5.9.

5.4.2.5 Creating templates for existing software platforms

This important activity is the only to be performed by somebody with conventional programming skills (i.e. programming using generic programming languages, e.g., Java). Considering that the overall semantics of the learning scenarios is ultimately defined by the generated software constructs, to more expressively represent concepts involved in the different leaning scenarios we will need a rich collection of templates mapping to the different software platforms. For this purpose, the MDEduc prototype comes with templates allowing to generate web applications.

However, once available, these templates allow the creation of learning scenarios that do not require general programming skills. Instead, it will require only the high level programming skill for the specific vocabulary concerned - an activity that dispenses with the costly intervention of regular programmers and that can be performed by learning designers.

5.4.2.6 Other use cases

The development of new learning models is considered a piecemeal process, which gradually unfolds full-fledged structures from an initial foundation by continual addition, improvement, repairing, etc. New learning scenarios will reflect such improvements, but existing ones should adapt to these evolutions. Here again, model transformations techniques and tools like the Merlin Generator can be used to make existing scenarios evolve from a given learning model to another version of this very same learning model.

There are also many other advanced use cases, which I will not consider in this thesis --like debugging the transformations, testing the generated applications, etc. -- but which can be equally important for the development process.

Ensuing sections will describe how Eclipse and EMF have been used to implement the MDEduc. It is highly advisable that readers not familiar with both Eclipse and EMF technologies consult the technical details of both frameworks in the appendices A and B of this dissertation.

5.5 The Pedagogical Pattern Editor and Model

There are no "typical" implementations for editors, because editors usually provide model-specific semantics. However, in the Eclipse framework, we can classify editors into two categories: text-based and non-text-based. Text-based editors can either inherit from the existing default text editors (e.g. `StatusTextEditor`, `AbstractDecoratedTextEditor`, etc.) or be created from scratch, using the facilities provided in the platform. If the editor is not text based, then they have to be implemented from scratch, in a plug-in of its own, according to the look and behavior desired (e.g. Form-based, List-oriented, etc.). PPEditor is an example of a text editor. Like the JavaEditor, PPEditor stems from `AbstractDecoratedTextEditor`, which means that it automatically inherits features like rulers, line numbers, quick diff, find/replace, etc.

Since MDEduc's Pedagogical Pattern Editor is text-based, I will concentrate on this kind of editor. Once the implementation model for the editor has been determined, implementing the editor is much like programming a stand-alone JFace or SWT application. Platform extensions are used to add actions, preferences, and wizards needed to support the editor. But the internals of the editor are largely dependent on your application design principles and internal model.

5.5.1 Parts of a text editor

The PPEditor is loaded whenever the user opens a file with the "*.pp" extension. In this case, the input to the editor is an `IFileEditorInput`. The platform text framework assumes little about the editor input itself. It works with a presentation model, called an `IDocument`,

which stores text and provides support for line information, text manipulation, document change listeners, search, etc.

The mapping from an expected domain model (the editor input) to the document is defined in an `IDocumentProvider`, which performs a few important tasks:

- maps editor inputs onto documents and annotation models
- tracks and communicates changes to the editor inputs into editor understandable events (`IElementChangeListener`)
- translates changes of the documents and annotation models into changes of the editor input (save)
- manages dirty state, modification stamps, encoding
- provides uniform access to editor inputs and their underlying elements

The MVC pattern is a constant in the Eclipse framework. In this case, SWT widgets perform the View part and JFace viewers perform the Controller part, acting as a bridge between low-level SWT widgets and domain objects (i.e. the Model part). This structure is also observed in the text editor framework, where `IDocument` performs the role of the model, the `StyledText` widget - a customizable widget that can be used to display and edit text with different colors and font styles - is the View, and `TextViewer` or `SourceViewer` (when, as it is the case of the PPEditor, the application requires features of source code editors) is the controller. Additionally, in an idiosynchrasy of the Eclispe platform, `TextEditor` (for example) is the context in which the different parts will be assembled. The `SourceViewer` is also responsible for the configuration of various aspects of an editor – like syntax coloring, content assist, hovers, formatters, etc -, a task that it delegates for a another class, the `SourceViewerConfiguration`.

Furthermore, since some operations (like *spell checking*, for example) may take a long time to run, PPEditor ascertains that they are run in a separate thread, the `Reconciler`, or they would likely impact on the interface's responsiveness. Figure 38 shows how some of the different parts used in the PP Editor fit together:

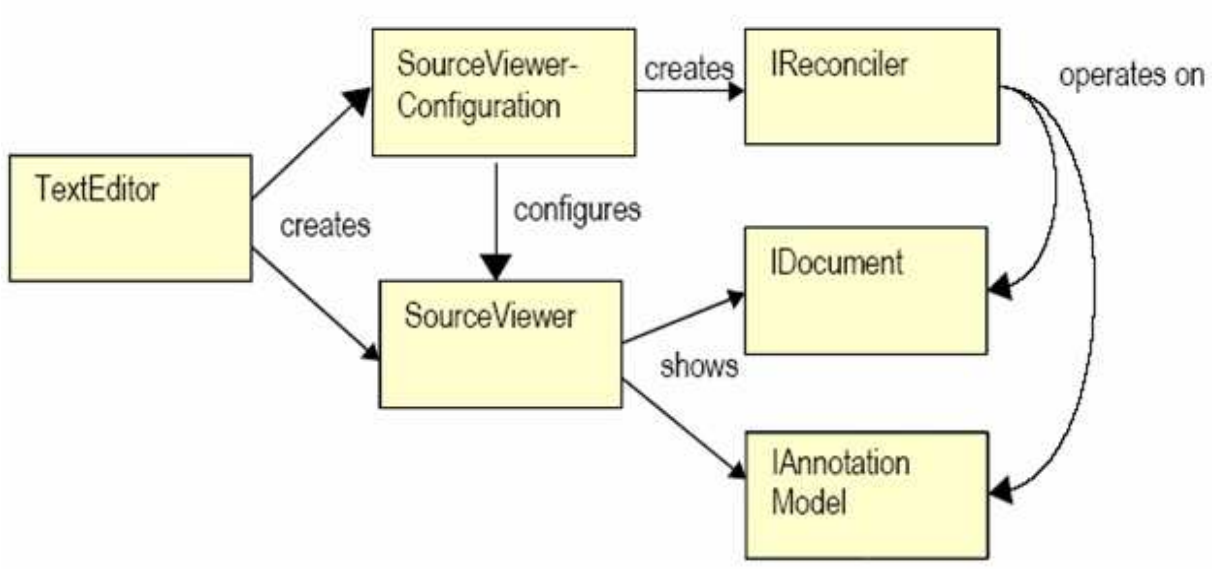


Figure 38: Basic structure of the PPEditor

5.5.2 Syntax Coloring

Concerning the syntax coloring, PPEditor maintains four different types of rules, namely, *single line* rules, *multi-line* rules, *word* rules and *concept* rules. The first three rules are conventional and appear regularly in text editors – in fact they are almost self-explaining, as we can see in Listing 9:

Listing 9: rules for syntax coloring used in the PPEditor

```
SingleLineRule linkRule= new SingleLineRule("[", "]", new Token(new TextAttribute(red, null, SWT.BOLD)));
MultiLineRule forcesRule= new MultiLineRule("Therefore,", ".", new Token(new TextAttribute(null, null, SWT.BOLD)));
keywordRule.addWord("Title:", new Token(new TextAttribute(null, null, SWT.BOLD)));
```

Thus, the above rules tell us, for example, that strings starting by "[" and finishing by "]" and not spanning more than one line will be colored in red (as we showed in Figure 31, for example, for the case of *links* to other patterns); likewise, strings starting with a "Therefore, " and finishing by a ".", possibly spanning multiple lines, will be automatically made bold (this rule is applied for the beginning of the solution section of a pattern.

Of course these rules alone, don't mean much if we don't specify in which "context" they apply. These "contexts" are what we call a *partition,* in the text editor jargon. In the text editor framework, a document is usually divided into non-overlapping regions called partitions, which are useful for treating different sections of the document differently with respect to features like syntax highlighting or formatting. So, for example, PPEditor creates a partition dedicated to *comments*, another for the pattern *Solution* section, etc.

The last type of rule, the *Concept rule*, on the other hand, is less conventional. It is responsible for "painting" the terms from the "domain" model that appear in the pedagogical pattern (e.g. domain's concepts in red, concepts' attributes in blue, etc.), where the "domain" model represents the learning scenario being designed. Contrary to the other rules, it applies to "dynamic" targets that are created at runtime. The excerpt below (Listing 10) shows that the model is read directly from an EMF Resource, whose content is iterated over; for each class it finds, a new word rule is created that paints the matching elements – in both capitalized and lower cased forms - with red.

```
private void getConceptWords(WordRule conceptRule){
      ResourceSet resourceSet = EduModelPlugin.getSharedResourceSet();
       for (Iterator iter = resourceSet.getResources().iterator();iter.hasNext();){
            Resource res = (Resource) iter.next();
            EPackage ePackage = (EPackage)res.getContents().get(0);
            for (Iterator i =  ePackage.getEClassifiers().iterator(); i.hasNext(); )
                {
                  EClassifier eClassifier = (EClassifier)i.next();
                  if (eClassifier instanceof EClass)
                      {
                         EClass eClass = (EClass)eClassifier;
                         Color red= fSharedColors.getColor(new RGB(250, 0, 0));
                         conceptRule.addWord(  eClass.getName(),  new  Token(new  TextAttribute(null,  red,
SWT.BOLD)));
                         conceptRule.addWord(        eClass.getName().toLowerCase(),        new        Token(new
TextAttribute(null, red, SWT.BOLD)));
                          for (Iterator j = eClass.getEStructuralFeatures().iterator(); j.hasNext(); )
                            {
                                  ….. //check if it is an attribute, reference, datatype, etc
                            }
            }
}
```

Listing 10: concept rule (excerpt)

5.5.3 Connecting the editor to the workbench

In order to connect a text/source editor with the rest of the Eclipse workbench, an *editor action bar contributor* is provided in which actions can be registered, enabling the access to other workbench features such as context menus, menu bars, and tool bars (please refer to Appendix A, for an explanation of the different Eclipse components). Editor action bar contributors are shared on a per editor *type* base, which is useful for code reuse, in case similar editors need the same actions, for example. The editor is informed of the existence of its action bar contributor declaratively, by the `contributorClass` parameter of the editor extension-point. In addition, the plugin.xml manifest file declares several other important attributes of an editor, like the file extensions associated with the editor, the editor's unique identifier, etc. (please see Listing 11).

```
<extension
      point="org.eclipse.ui.editors">
  <editor
   class="org.ppeditor.PPEditor"
   contributorClass="org.ppeditor.PPEditorActionContributor"
   default="false"
   extensions="pp"
   icon="icons/pp.PNG"
   id="org.ppeditor.PPEditor"
   name="Pedagogical Pattern Editor">
  </editor>
</extension>
```

Listing 11: Declaring an editor in the plug-in manifest file

A feature that greatly enhances file navigation is the synthetic content outline. In the case of the Java Editor, for example, the provided outline is a syntax tree restricted to the Java model (i.e. fields, methods, internal classes, etc.). Therefore I decided to implement a *content outline page* to be displayed in the Content Outline view. Also, we can consider that implementing a content outline page is another way of connecting an editor to the workbench. For this to be possible, a second plug-in has been created containing the classes that implement the pedagogical patterns model (i.e. *link*, *alias* and *concept*) as well as a class that parses the text from pedagogical patterns to create instances of the model elements. Figure 39 illustrates this plug-in.

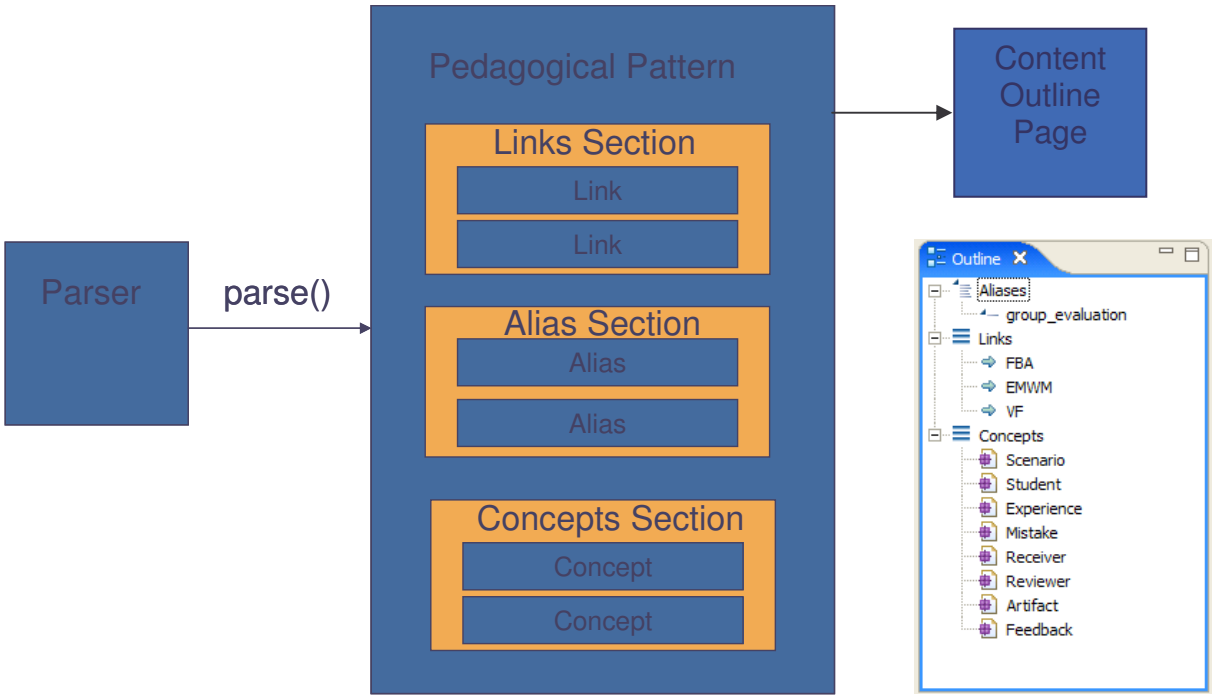


Figure 39: Parsing a PP Model (left) and resulting outine view (right)

The PPEditor participates in the content outline presentation by declaring among its adapters an IContentOutlinePage, which the Outline View uses to extract the information it displays. The implementation of this interface must offer the needed information in the convenient format, which means that it must implment the ContentProvider and the LabelProvider interfaces. Thus, PPEditor now knows of its outline page and updates it on every input change.

A question that comes up at this point is “how to achieve the synchronization between the text, the content outline view and the different ‘models’ (i.e., the pedagogical pattern model and the domain model)?” First of all, there are two mechanisms involved: the first is the “reconciling” mechanism of the platform text framework and the second is based on the notifier/observer pattern, implemented via Eclipse’s extension point. While the first mechanism will be explained next in this section, the second one will be addressed only in the section 5.8, which deals with the synchronization between plug-ins.

Recognizing that the task of constantly refreshing the presentation would be too heavy to be executed in the UI thread, MDEduc implements a "Reconciler", an object that defines and maintains a model of the content of the text viewer's document in the presence of changes applied to this document. As mentioned above, the reconciler runs the analysis in a separate

thread that is triggered whenever the user stops typing and that is interrupted when the user types again. In this way, the content outline view is able to refresh the pedagogical pattern elements it displays, making them always synchronized with the text being edited.

5.6 The EduModel

As advanced in Section 5.3.2, EduModel is the piece of software that will help designers to extract formal languages from informal ones. Essentially, it leverages the EMF dynamic framework for building metamodels (i.e. learning scenarios) based on the Ecore meta-metamodel.

As previously observed, the EduModel does not have an editor or view of its own, since it relies on the parts[42] used by the informal description. However, it contributes its own actions (please refer to Figure 33), accessed through a context menu, where the creation of the model elements actually takes place. The actions provided by EduModel are:

- **New Link**: Creates a link (an `EReference`, in the Ecore jargon) between two concepts from the term selected in the informal description representing the learning scenario.
- **New Property**: Creates a new property (an `EAttribute`, in the Ecore jargon) for a Concept from the term selected in the informal description representing the learning scenario.
- **New Concept**: Creates a new Concept (i.e. an Ecore `EClass`) from the term selected in the informal description representing the learning scenario.
- **New annotation**: Allows to annotate existing constructs (e.g. an existing Concept). This can be used to provide semantic information (i.e. to create *stereotypes*, as it will be explained in the next section) to the annotated construct or just to add extra relevant information about it. It's implemented using the Ecore `EAnnotation` class.
- **Associate Model**: Associates an existing Ecore model with the informal description being edited or visualized.

[42] In the Eclipse workbench, a part is a view or an editor.

Corresponding to each action above, there is a dialog demanding further information about the elements to be created, as we can see in Figure 40 for the case of the "New Link" action (note that the element's name has already been filled out through the selection mechanism):

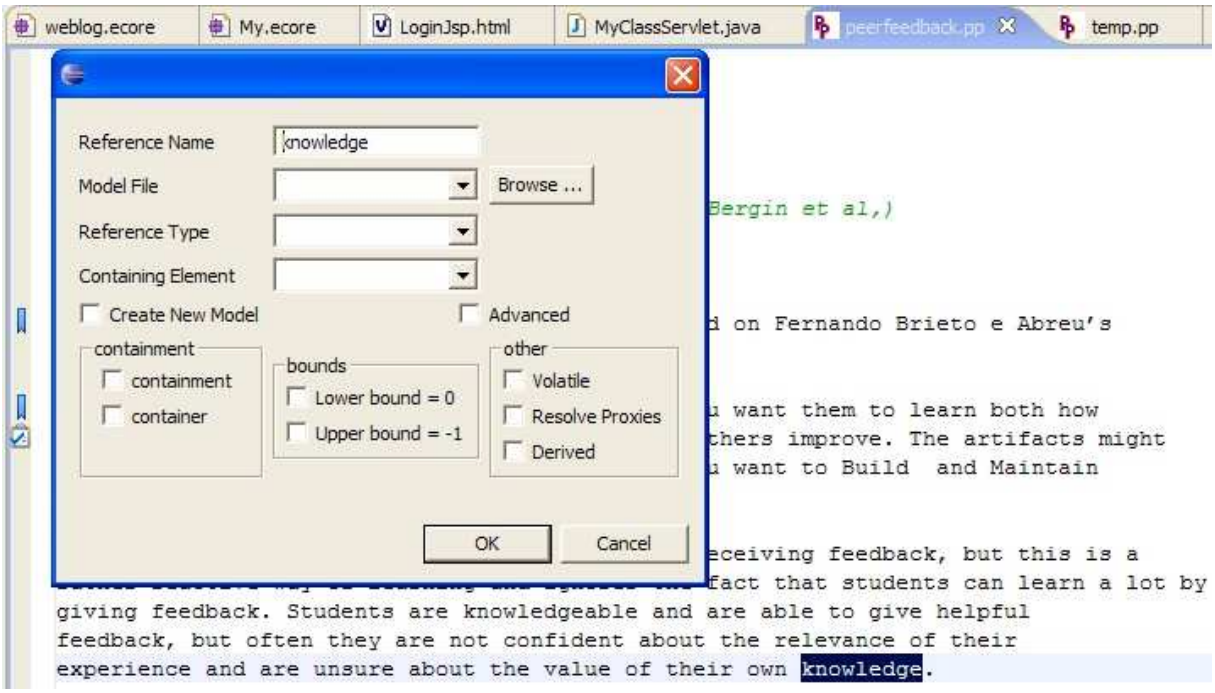


Figure 40: Creating a new link with the EduModel

After some interactions creating concepts, attributes and other elements of the model, an Ecore file is created that expresses the same learning scenario described informally by the textual notation that guided its creation (a pedagogical pattern, for example) , *as viewed by the modeler*. An example of such a file has been provided in Figure 35, for the Peer Feedback metamodel. That is, certainly the steps taken, the concepts chosen, etc. will be a result of a subjective interpretation of the designer vis-à-vis the original source and is highly likely that first versions of models will be revisited several times before designers feel pleased with it.

Nevertheless, and this is one of the main advantages of the LOP paradigm, we don't have to wait until we are sufficiently satisfied with the model, before we can create an application realizing it. On the contrary, it is by generating and executing applications that we can see how the model works, obtaining further information leading to its improvement and ultimately validating it.

What it is important is that, once we have an Ecore model, we enter the "EMF world" and consequently can take advantage of everything it has to offer: code generation, model

transformation, and so on. Furthermore, since models created are conformant to the same meta-metamodel (i.e. Ecore), they all know about each other, and can potentially be interconnected, fusioned, *inter alia*.

5.7 The EduGen

With the help of Figure 41, we can see how EduGen relates to EduModel and how both fit in the MDEduc framework. The figure makes use of the MDA nomenclature (i.e. M3, M2, M1 and M0 layers) to illustrate how the two applications, starting from a meta-metamodel (i.e. Ecore), enable designers to create a domain model – the last step, i.e. the creation of the executable application tailored to the domain model is done by the Java virtual machine.

As seen before, the EduModel makes use of the Ecore meta-metamodel constructs to create the domain metamodel (or the Domain Specific Language), which in the case is the Peer FeedBack DSL. In this DSL, abstract and concrete syntaxes and grammar for peer feedback scenarios are proposed. EduGen, in turns, takes over from the EduModel to propose a specific way in which Peer Feedback DSL constructs can be arranged to implement a specific scenario, aimed at a particular platform.

A further complication, not explicit in Figure 41, is that EduGen, by using a templating mechanism (i.e. JET's), introduces concepts of the platform chosen to execute the Peer Feedback model in the process. In MDA terms this would mean that EduGen "transparently" (at least from the learning designer perspective) introduces the Platform Description Model (PDM). That is, EduGen executes a "fusion" between the PIM (i.e. the Peer Feedback model) and the PDM (e.g. a web application), though the latter is not actually a model but rather a set of "hard-coded" templates.

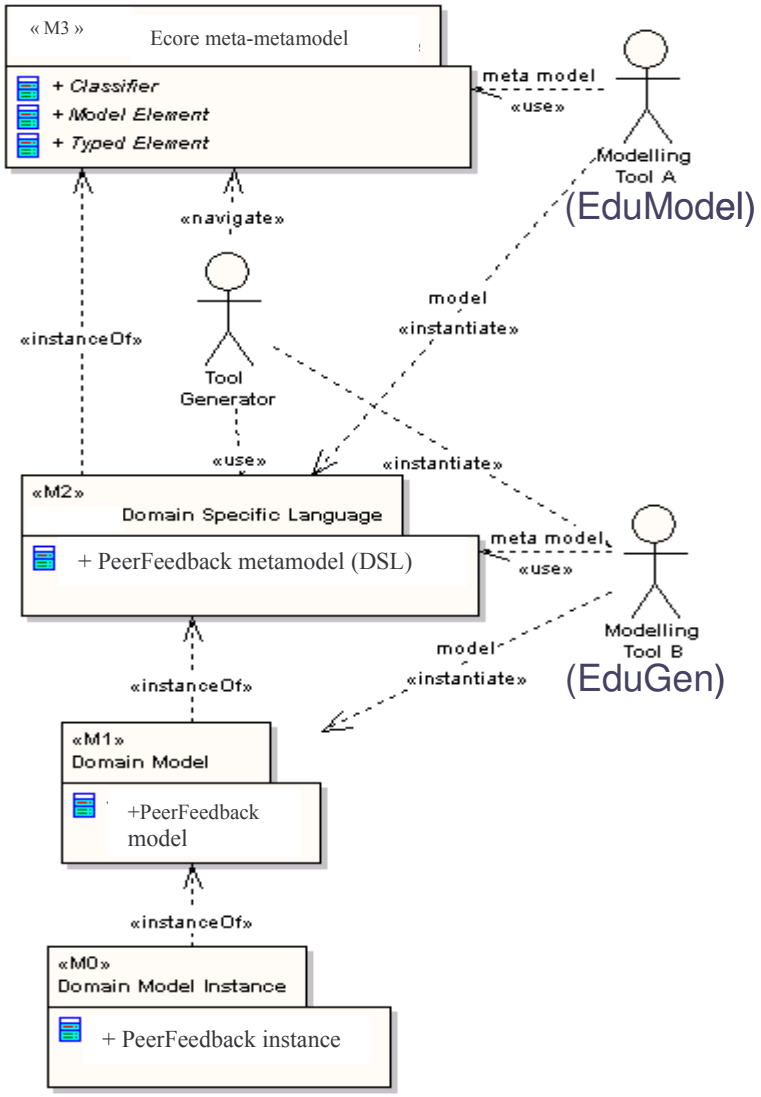


Figure 41: EduModel and EduGen related (source: adapted from a diagram by Jorn Bettin, sent to the GMT discussion list)

In order to choose the « good » template to apply, EduGen makes also use of UML stereotypes (Fowler, 2004). Since the Ecore meta-metamodel does not present the notion of a stereotype, it has to be emulated with Ecore's `EAnnotation`. In that case, for each `EClassifier` in a given model, an `EAnnotation` sub-element is created, and, each time this classifier is passed in as the template's *argument* variable, GenModel checks to see what kind of annotation it has and takes the right action accordingly. Listing 12 shows how this checking is done.

Listing 12: Using Ecore `EAnnotation` to simulate UML stereotypes

```
public static final boolean hasStereotype(EClassifier eClass, String stereoType){
  EAnnotation eAnnotation = eClass.getEAnnotation(UML2_MODEL_PACKAGE_NS_URI);
  if (eAnnotation != null)
    return stereoType.equals((String)eAnnotation.getDetails().get("stereotype"));
  else return false;
}
```

Additionally, GenModel provides « service » classes that help keep templates simple. These service classes offer APIs to more conveniently deal with Strings, Ecore elements, JET calls, etc, trimming down the amount of code required in the templates' scriptlets and expressions. For example, the method `hasStereotype`, listed above, belongs to the *Ecore services*, which means that if we want to write a template that checks for a given stereotype, we do not have to reproduce the code from Listing 12, needing only to write something like:

Listing 13 : Using service classes in a JET scriptlet

```
<%if(EcoreServices.hasStereotype(eClass,"Entity")){ %>
```

5.8 Other MDEduc features

5.8.1 The Browser View

So as to allow learning designers to create formal models out of existing learning scenario descriptions found on the Internet, the MDEduc provides also a special browser view. It is basically an Eclipse view (i.e. extends the `ViewPart` abstract class) which, to the basic features present in every browser, it adds two others so it can seamlessly integrate to the rest of the workbench:

- **The Browser view must be a** `SelectionProvider`: This ability, whose implementation is trivial in ordinary views and editors - like the PPEditor – is not so easily accomplished in the browser view. The problem arises because the SWT browser widget does not surface selection events, since underlying OS resources (i.e. the different browsers) do not publish such events in a standard way. Thus, the workaround used was

to have the selections captured through a JavaScript program and forwarded to the browser widget. So, whenever a new page is loaded, a `ProgressEvent` is generated that fires a javascript code that captures the selection in a string and sends it to the `StatusLine` (the browser's status bar). When the status line receives this selection it throws a new event (`TextSelection`), finally an event recognizable by the workbench.

- **It must accept the EduModel context menu**: In this case, instead of presenting the regular OS provided context menu (which depends on the browser, e.g. Firefox, Internet Explorer, etc), the browser view overrides it with the context menu required to build models (e.g. "New Link", "New Concept", etc.).

Physically, the Browser View is located in the EduModel plug-in and it uses the SWT Browser widget, which serves as a façade to external browsers.

5.8.2 The MDEduc Perspective

An Eclipse *Perspective* is a layout of editors and views that reflects a certain task (e.g. debugging, coding, accessing resources, etc.), molding the workbench correspondingly so the user may be able to perform the task more easily. MDEduc provides its own perspective, the *MDEduc Perspective*, which privileges the modeling task.

By default, the MDEduc Perspective includes the two MDEduc parts, i.e. the *PPEditor* and the EduGen's *Browser* view, the *Navigator* view, which enables the access to both created and generated resources (models, manifest files, classes, etc.) and, finally, the *Content Outline* view, which facilitates the navigation through the different resources. Of course, this is just a suggestion of layout and can be reconfigured at any time by the user - for example adding or removing views - if needed. Figure 42 shows the default layout of the MDEduc Perspective:

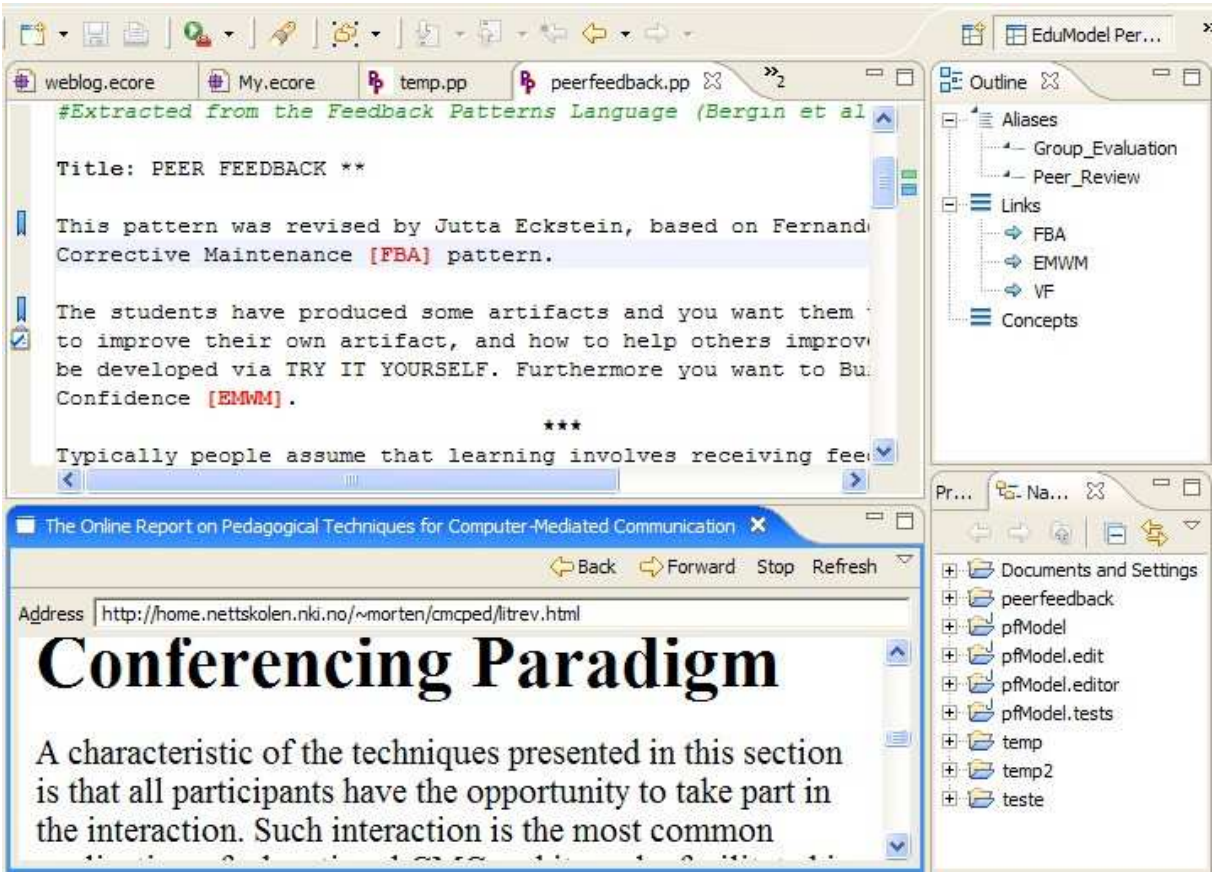


Figure 42: the MDEduc Perspective

5.8.3 Synchronizing plug-ins

It should be clear by now that MDEduc is composed of four plug-ins that work together to offer the functionalities required by the design of learning scenarios (both formal and informal) and by the subsequent generation of code. From the previous sections, it should also seem obvious that some mechanism of synchronization is required that allows the plug-ins to work independently of one another while enabling each plug-in to be notified of changes committed into the others.

Also, this synchronization should be achieved without coupling the plug-ins to one another, permitting each one to be expanded without interfering with the others. In such situations it is common to resort to the observer design pattern (Erich Gamma et al., 1995), which defines "a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically". It is widely used when we want to decouple a model from its presentation(s), as it is exactly the case of MDEduc.

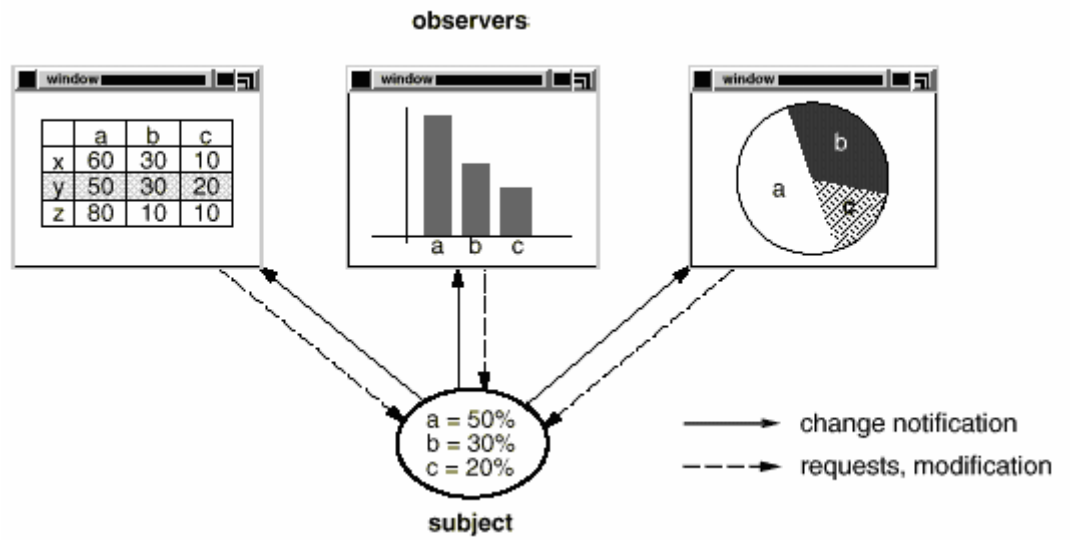


Figure 43: Observer design pattern (source: Erich Gamma *et al*., 1995)

Thus, two interfaces have been designed to represent, respectively, the subject and the observer: the IModelChangeProvider and the IModelChangeListener. The "provider" (i.e. the subject) part presents a method that lets "listeners" (or observers) to manifest their interest for the change in the subject (in MDEduc's case a change in the model). In turn, observers offer a standard method that will be called by the subject to notify that a change has occurred; this method will be called once for each instance in the subject's observer stack. This is represented in Figure 44.

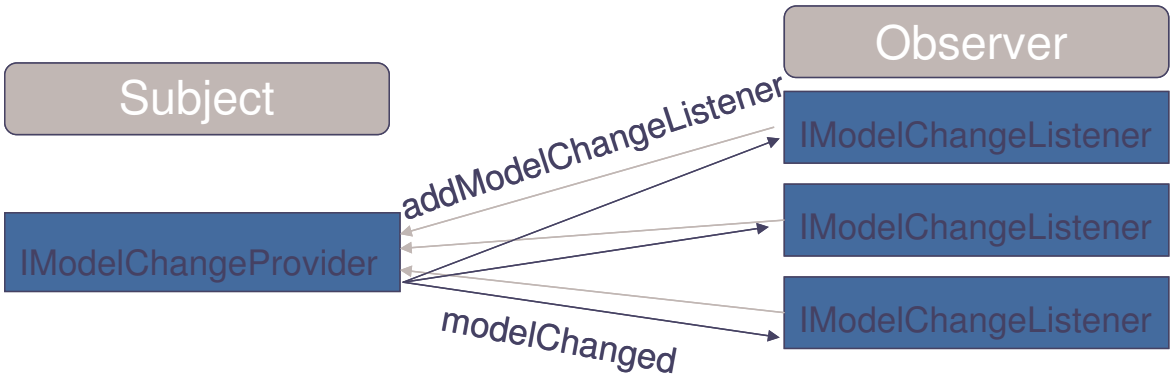


Figure 44: Dynamic listener observer in MDEduc

Certainly these two interfaces should make part of the plug-ins' public APIs. Nevertheless, particularly the Listener interface is, *par excellence,* a point to be extended in its implementing plug-ins and, consequently, more than just public, its methods should be made published, as explained in Section A.3.1 of Appendix A. This means that the Listener

interface should be declared via the Eclipse's extension point mechanism, facilitating the task for plug-ins extending it.

However, if declaring extension points make the life easier for developers wishing to extend a given plug-in, it is not exactly the same for developers that want their plug-ins to be extended For each extension point they want to provide, they have to:

- Design the extension point;
- Define a schema file: It is in this file that the extension point is declared (specifying details, like required parameters, etc);
- Define the extension point in the plug-in manifest file, plugin.xml;
- Write code to load the extensions to that point: i.e. they have to code the Java classes that are used at runtime to parse the contributed *Extension*s and store all their associated data (`IConfigurationElements`);
- Invoke the extensions (safely) at the appropriate time: i.e. the code mentioned in the preceding item should be given a special concern to avoid being broken by malicious extenders.

The complication is further aggravated by the fact that Eclipse "only provides a generic interface to query the extensions and that there is no standardized way to query the original extension point schema at runtime" (Stepper, 2006). Anyway, once the items described above are implemented, the developer can make use of the facilities offered by Eclipse PDE framework to build an "extender" plug-in and, by defining an *extension,* be guided through the development of the part of the plug-in that implements the extension.

As we see in Figure 45 (on the left side), in the case of the MDEduc, the `listeners` extension point is defined by the EduModel to allow other plug-ins (for example, the PPEditor) to be notified of any change made to the model. For example, a new concept has been added to the Peer Feedback model and this should reflect in the PPEditor – in this case, the Peer Feedback pedagogical pattern being edited will present all the words corresponding to the concept changed into red.

As defined in the Listeners.exsd extension point schema (whose general structure can be seen in the screenshot of the Content Outline view in Figure 45, right), the *listener* element

contains, among others, an important sub-element called `class`. This extension declaration, part of which is reproduced in Listing 14, says that when an extension point named `listeners` is invoked, an instance of `eduModel.IModelChangeListener` should be created that must handle the event (in the case of the PPEditor, this is described in the `modelChanged` method).

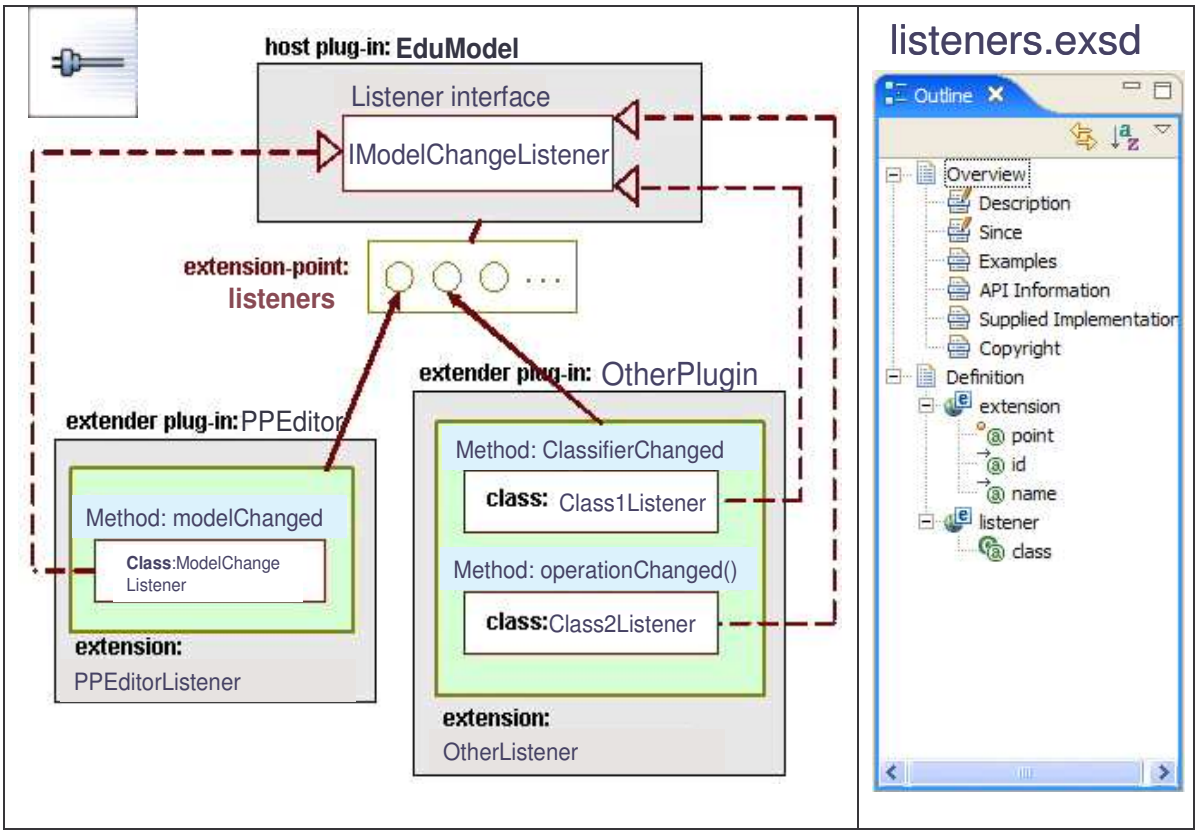


Figure 45: Notification via extension point (left) and outline of the extension point schema definition (right)

```
<element name="listener">
 <complexType>
  <attribute name="class" type="string" use="required">
   <annotation>
    <appInfo>
    <meta.attribute kind="java" basedOn="eduModel.IModelChangeListener"/>
    </appInfo>
   </annotation>
  </attribute>
 </complexType>
</element>
```

Listing 14: `listener` extension point schema definition

5.9 Developing the "What is Greatness" scenario

Since the main mechanisms of the four MDEduc plug-ins have been scrutinized in the last section, in this section I will illustrate their usage in a concrete learning scenario. For this example, the whole design process is:

- The scenario author (an instructor, for example) edits an informal description of the learning scenario;
- Taking over from the scenario author, the learning designer uses the EduModel to elaborate a formal model (serialized in an Ecore file).
- Then, the learning designer uses the EduGen plug-in to generate the final code, a web application using the Java JSP technology.

For this example, I decided to use the "What is Greatness" scenario (Dalziel, 2003), since it has already been implemented for different platforms, and more notably for the LAMS platform (Dalziel, 2003) and the IMS-LD specification. At the end of the process (which for simple examples like this one, should not take longer than a few minutes), a web application conformant to Dalziel's scenario will be generated (Figure 46 shows the screenshot of one of its interfaces).

Figure 46: generated web application for the "What is Greatness" learning scenario

5.9.1 Creating the "What is Greatness" metamodel

An excerpt of this scenario textual description can be found in Figure 47. Because this initial description is not in the notation of pedagogical patterns, it does not make sense to begin this example with the PP Editor and, since it can be retrieved directly from the Internet, the learning designer can use the browser explained in the previous section to capture the constructs of the "What is Greatness" metamodel. Figure 47 shows this step.

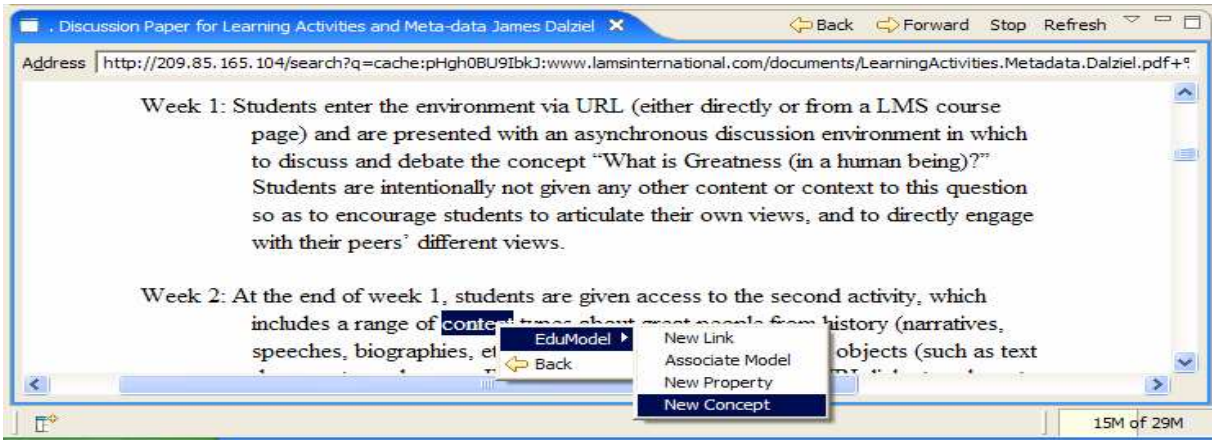


Figure 47: Using the browser to create the "What is Greatness" model

In this particular case, knowing in advance that the metamodel will be mapped to a web application, the learning designer decided to separate the different concepts into two main packages: one reserved for *entity-like* concepts (grouped in the **entity** package) and other to keep the *process-like* concepts (grouped in the **session** package). Certainly, this is a completely arbitrary design decision, and serves only to keep the example more organized, having no significant consequence on the generated data. Of course, this kind of consideration is not expected from all learning designers.

For "entity-like" I mean concepts that basically serve to store values and that will have a counterpart in a database. In that case, each of these concepts will generate a Data Transfer Object (DTO) and a Data Access Object (DAO)[43], which will mediate the access to the database. Examples of entity-like concepts are **Student** and **Teacher**.

In turn, "session-like" elements correspond to the user interface and to the controller of web applications, being consequently mapped to two JET templates: one giving rise to a JSP and another to a servlet. The session-like concepts represent the activities to be performed by the participants and, also for the sake of tidiness, have been further grouped into the packages **students** and **teachers**. These mappings can be better visualized in Figure 48.

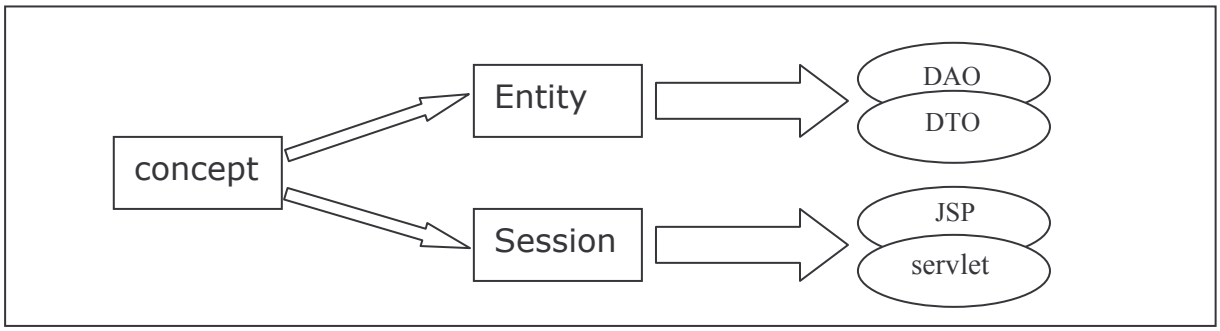


Figure 48: mapping model concepts to web application constructs

After a few interactions creating the different concepts, the metamodel illustrated in Figure 49

[43] DAO and DTO are two design patterns that make part of the Sun's Core J2EE Patterns catalog (online at http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html)

is created. In fact, Figure 49 represents just a simplified tree-based notation of a complete specification saved to a file and expressed in the XMI idiom. The XMI file of this example is reproduced in Annex D. This file can be accessed using the EduModel's **Associate Model** action, which prompts the user for a file name (if it already exists, the model's constructs will be saved to this file; if the file entered by the user does not exist, a new one will be created).

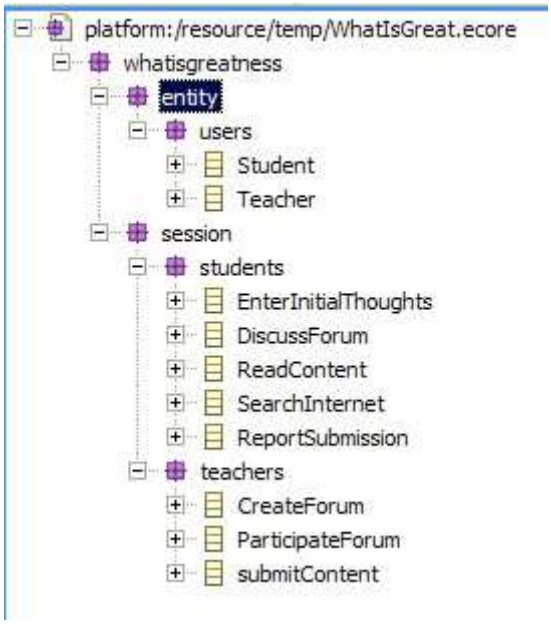


Figure 49: What is Greatness metamodel

5.9.2 Mapping metamodel concepts to templates

A critical aspect is to decide how to implement the mappings between elements of a metamodel and the different templates. For example, where has it been decided that **student** maps to entity-like software artifacts (i.e. to DAOs and DTOs)? As explained in Section 5.7, the EduGen approach is to insert this information in the form "stereotypes" (which in Ecore based metamodels are implemented via `EAnnotations`). This means that the model elements will have to be annotated with the desired stereotype (in this specific case, either *entity* or *session*).

However this is just one possible solution. In fact, even if this approach presents the advantage of concentrating everything needed to generate applications in the same place (i.e. in the learning model itself), it presents some limitations:

- it "pollutes" the learning model with information related to the platform;

- it requires some logic to be added to the templates in order to filter the concepts' stereotypes, which adds more complexity to the templates; for example, in this specific case, once it has been detected that Student is an entity-like concept, only templates of DAO and DTO will produce some output.

Another possibility to performing these mappings is by using an additional file, serving to keep platform related configurations. The EMF default implementation, for example, uses an extra metamodel (the GenModel), which basically decorates the business model with generation specific information. Also the Merlin Generator uses a similar approach, by requiring a mapping file to match metamodel elements to templates. Next version of the EduGen will also bring this feature.

Once this high level mapping is decided, another one, more fine-grained mapping between concepts' sub-elements (e.g. attributes, references, etc.) and the different parts of the generated software is required. In the EduGen, this is automatically accomplished by the templates. For example, for the case of the web application templates, while each session-like concept gives rise to a JSP and a servlet, each *attribute* of this concept (i.e. Ecore's `EAttribute`) generates an input field of a form in the JSP file and a field in the different methods of the servlet. This mapping basically reflects the fact that attributes in models and input fields in forms and fields in servlets serve to store values (i.e. to keep states). For example, for the attributes of the DicussionForum concept, this corresponds to the following excerpt:

Listing 15: Generated JSP for the DiscussionForum's attributes (excerpt)

```
<form action="<%=request.getContextPath()%>/discussforum/" method="POST">
        <input type="hidden" name="event" value=""/>
                <table>
                        <tr>
                                <td>lastName :</td>
                                <td><input type="text" name="lastName"/></td>
                        </tr>
                        <tr>
                                <td>firstName :</td>
                                <td><input type="text" name="firstName"/></td>
                        </tr>
                        <tr>
                                <td>email :</td>
                                <td><input type="text" name="email"/></td>
                        </tr>
                        <tr>
                                <td>login :</td>
                                <td><input type="text" name="login"/></td>
                        </tr>
```

Likewise, *references* of a concept (i.e. Ecore's `EReference`) serve to keep links to other objects, being semantically equivalent to JSP's links (i.e. the <a href> html tag) or to the servlet's strings referencing JSP pages to which responses will be forwarded (via request dispatcher objects). Corresponding generated codes for the "What is Greatness" example scenario can be seen in Listing 16 and Listing 17.

```
<li>
        <a href="<%=request.getContextPath()%>/readcontent/">
                <i>ReadContent</i> : great.doc</a>
</li>
<li>
        <a href="<%=request.getContextPath()%>/enterinitialthoughts/">
                <i>EnterInitialThoughts</i> : initial thoughts</a></li>
<li>
        <a href="<%=request.getContextPath()%>/searchinternet/">
                <i>SearchInternet</i> : http://google.com</a>
</li>
```

Listing 16: Generated JSP for the references of the DiscussionForum concept (excerpt)

```
public final static String PAGE_READ = "readContents.jsp";
public final static String PAGE_CREATE = "createInitialThoughts.jsp";
```

Listing 17: Generated JSP for the references of the DiscussionForum concept (excerpt)

Finally, to give but three examples, an *operation* in a concept (i.e. Ecore's `EOperation`) is semantically akin to JSP's forms with submit buttons (one button for each operation to be designated after the operation's name) and to the servlet's methods called when an action is requested by the user (these methods are called from the servlet's `doGet/doPost/doAction` standard methods). An excerpt for the login `EOperation` of the DiscussForum concept is reproduced in Listing 18.

```
public void login(HttpServletRequest req, HttpServletResponse resp) throws
                                        ServletException, IOException {
        log.debug("Starting DiscussForum.login");
        String lastName = (String)req.getAttribute("lastName");
        String firstName = (String)req.getAttribute("firstName");
        String email = (String)req.getAttribute("email");
        String login = (String)req.getAttribute("login");

        RequestDispatcher dispatcher = req.getRequestDispatcher("/jsp/"+ PAGE_READ);
          if (dispatcher != null) {
              dispatcher.forward(req, resp);
          } else {
               throw new ServletException("failed: " + PAGE_READ + " was not found");
          }
        }
```

Listing 18: Generated servlet for the operations of the DiscussionForum concept (excerpt)

By the way, Listing 18 also shows the generated code for the attributes of DiscussionForum (i.e. for each attribute, a variable has been created). While there are several other mappings in the different templates, these three ones are sufficient to illustrate the generation of a JSP file, described next.

5.9.3 Generating the "What is Greatness" web application

The rest of the model must be completed, i.e. attributes, references, operations, etc. have to be added to the core "What is Greatness" metamodel and their properties have to be filled out with actual data (using the Eclipse Properties view). Once the model is complete, all we have to do is to choose the **Generate** action of the EduGen plug-in (it is a context menu enabled for Ecore files in the Navigator view). Then, a dialog will appear prompting the user for the desired *platform profile* (in fact a folder specifying where the platform's templates are) and the location where the software will be generated. The only available platform in this prototype version of the EduGen is a simple web application.

After the generation process (which should not take longer than a few seconds), a web application will be created that is conformant to the *What is Greatness* metamodel. Figure 50 shows the session-like DiscussForum concept and Figure 51 reproduces the screenshot of the Eclipse interface after generation process. Please note that the Navigator view shows the generated programs (JSPs, servlets, HTMLs, etc.) corresponding to the different concepts of the web application and a browser view lays out the JSP automatically generated for the DiscussionForum concept. In particular, observe that, conforming to the three mappings detailed above, *lastName*, *firstName*, *email* and *login* `EAttributes` gave rise to input text fields in the JSP; *readContents*, *enterInitialThoughts* and *searchInternet* `EReferences` generated hyperlinks to other pages of the web application and the *login* `EOperation` yielded a form and a submit button with the same name in the JSP.

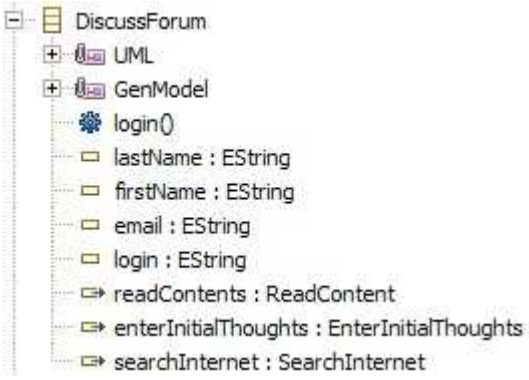


Figure 50 : The DiscussionForum concept (expanded)

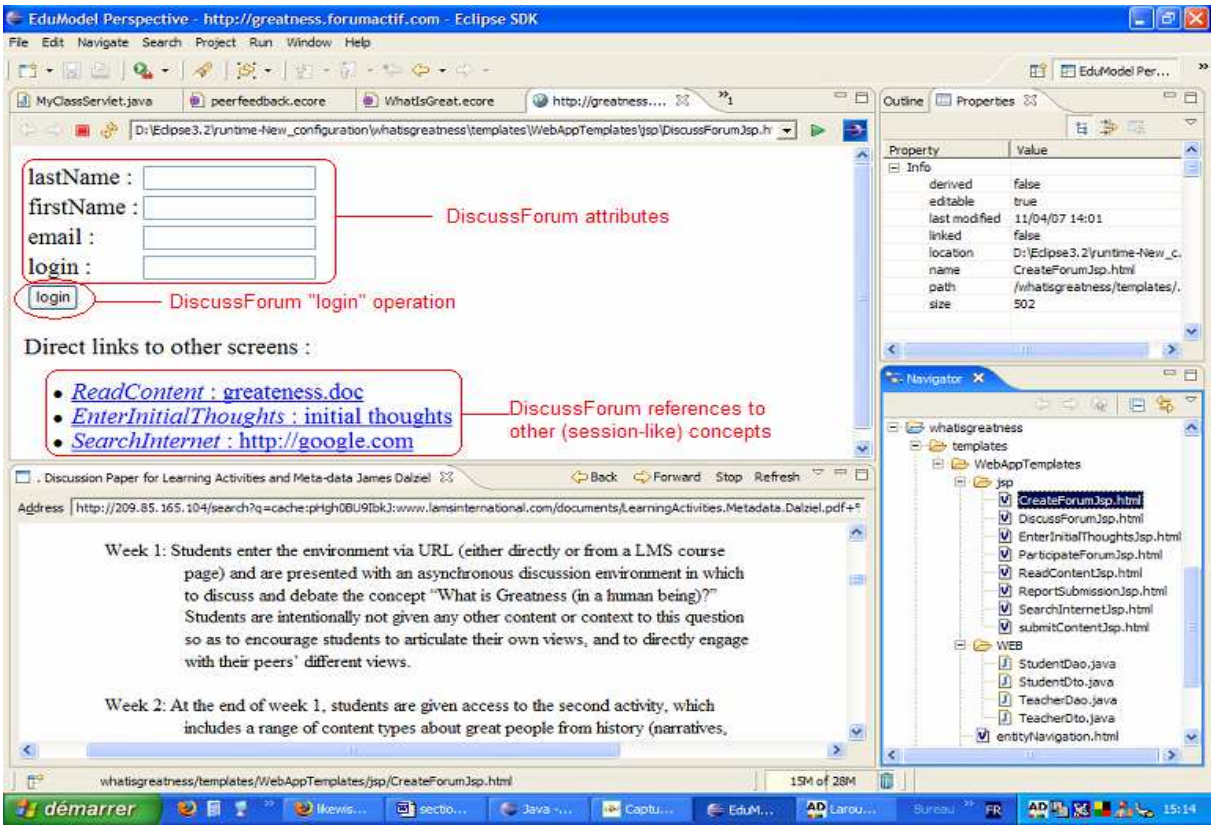


Figure 51: Generated web application for the "What is Greatness" scenario

To complete the web application, one deployment descriptor, one ANT build file and some navigational pages (one for each `EPackage`) are also generated. Thus, this example illustrates the whole cycle of development of a learning scenario. The Eclipse Navigator view (Figure 51 bottom right) shows some of the generated files of the whole web application.

5.9.4 Generating other applications

Only the metamodel and the templates are necessary to decide what application will be generated at the end. Considering that designers may come up with any metamodel and that templates can be created for any platform, the semantics of the learning scenario will be only restricted by the computer possibilities. For example, if someone decides that the discussion forum should be replaced by a simulation program and he/she provides the correct templates, then the semantics of the scenario will change accordingly. This contrasts with IMS-LD's approach, which offers fixed vocabulary and also fixed semantics (e.g. the *activity* concept is always implemented as a web form – i.e. a text with some text serving as guideline and input fields, like text field, drop down menu and a few others).

5.9.5 Transforming scenarios

Let us suppose that the learning designer, after having created the "What is Greatness" metamodel, decides to execute it in a standardized platform. In this case, the Eclipse Modeling Framework comes handy, by providing some tools that will permit the transformation between this metamodel and the target platform's metamodel.

For example, suppose that a learning designer wishes to execute the "What is Greatness" scenario in the Coppercore, the IMS-LD's runtime engine. The first step is to express the metamodel created above into the IMS-LD constructs. This metamodel transformation can be easily achieved using the Merlin Generator. By creating a mapping file in which the semantic correspondences between the concepts of both metamodels are laid out, the learning designer can subsequently transform instance scenarios of one into the other.

Figure 52 illustrates a toy example, in which the main concepts of "What is Greatness" and IMS-LD have been matched. We see for this simple example that session-like concepts of the "What is Greatness" metamodel have been mapped to the IMS-LD *LearningActivityType* and *SupportActivityType* concepts, while the entity-like concepts, *Student* and *Teacher* have been mapped, respectively to IMS-LD's *LearnerType* and *StaffType* constructs. Certainly, for real cases, a much more fine-tuned transformation should be provided, for example, by matching attributes, references, etc. and even adding transformation rules (for this, Merlin makes use of a Java-like scripting language).

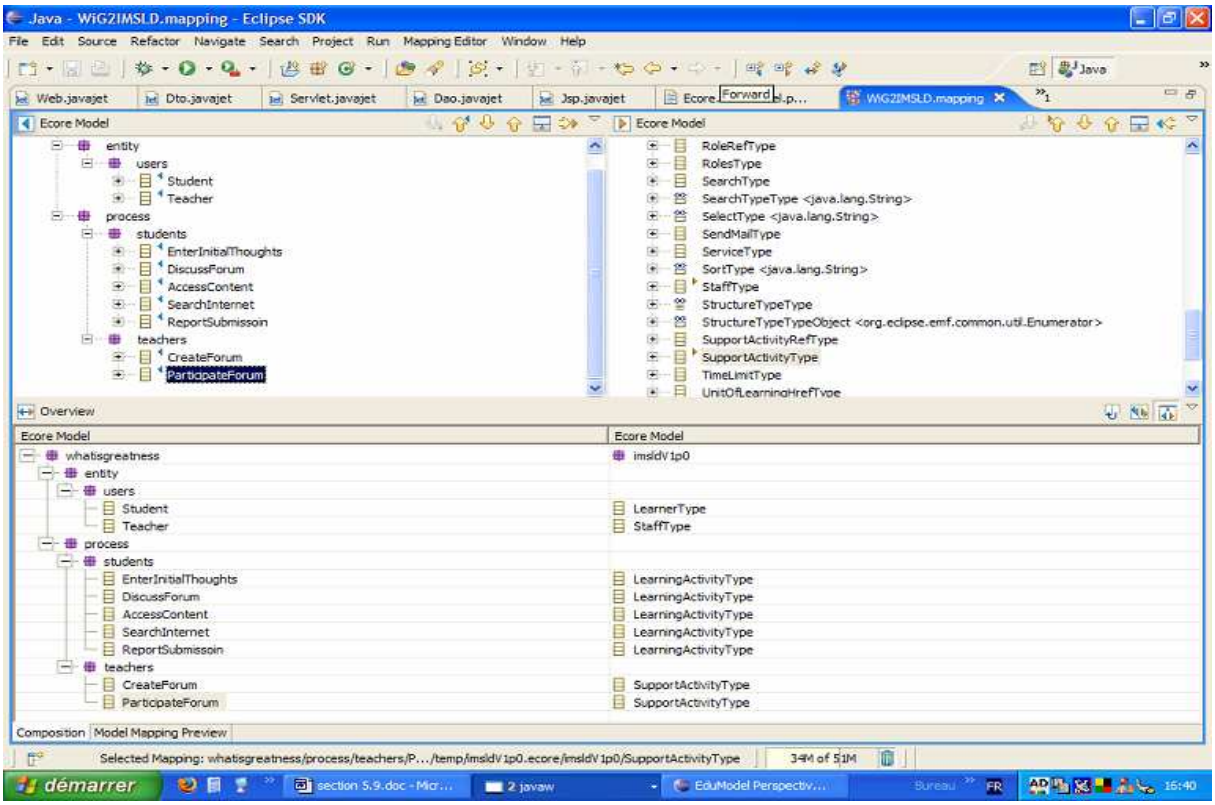


Figure 52 : Transforming between “What is Greatness” and IMS-LD metamodels

5.10 Conclusion

This chapter described some practical aspects of the MDEduc, a prototype that realizes some of the ideas advocated in this thesis. As we could see, MDEduc is an environment where education practitioners can use the PPEditor to describe their pedagogical strategies using textual notations expressed as pedagogical patterns, and where learning designers, using the EduModel, may take over from the educational practitioners to create a model from scratch and then to generate an application – as opposed to code it by hand - where the model can be edited and executed.

MDEduc has been conceived to facilitate the task of creation of learning scenarios. Nevertheless, I am aware that its usage may still be challenging for designers not having a technical background on theory of models, languages and some other subjects discussed in this thesis. Next chapter I will describe strategies on how to smooth out the path to both domain experts and learning designers wishing to investigate more deeply on the approach proposed by the MDEduc.

Chapter 6: Conclusion

> "To solve really hard problems, we'll have to use several different representations … [and] … this is because each particular kind of data structure has its own virtues and deficiencies, and none by itself would seem adequate for all the different functions" (Marvin Minsky).

6.1 Review

As discussed in the introduction of this dissertation, software applications designed for use in education are still in debt with their users, concerning tangible evidences that they represent a major advance in the development of educational tools to enhance learning. Aware of this fact, researchers rush into developing new approaches to help designers and developers conceive better learning applications.

Of the various aspects that can be researched, aimed at improving the quality of learning tools (e.g. how to develop more engaging tools, how to conceive more interactive interfaces, etc.), there is one that interested particularly this work: how to conceive applications that better reflect the pedagogical strategies of practitioners. The rationale is simple: since there are teachers that are more successful than others in motivating students, in instilling more long-term values, in exploring the potential of learners, and so on, all of these competences contributing to make learning easier, quicker, more enjoyable, etc, then if we somehow manage to capture their "teaching secrets" in the form of software programs, then maybe we would be able to develop more didactical tools, obtaining, as a consequence, better learning outcomes through using them.

Thus, this dissertation set out to discuss the possibility of designing software applications that captured the knowledge of domain experts in the area of education (teachers, instructors, learning designers, *pedagogues,* etc). Besides, because applications are only a part of a bigger picture, which I called a learning scenario, I focused on the broader question of designing automated learning scenarios.

More specifically, this thesis described a train of thought in the development of a novel approach to conceiving learning scenarios that offered an alternative to two existing techniques, namely, Instructional Systems Design (ISD) and Educational Modeling Languages (EML). In fact, while the resulting approach, called Multi-EML, draws on the latter, it criticizes its main assumption that an EML should be an integrative model, intended to represent every pedagogical strategy.

I maintained throughout this work that there is no model prevailing over the rest and that yields optimal results in all possible situations, as implicitly assumed by the proponents of the one-model-fits-all approach. Instead, I proposed a theoretical and a practical framework that, by acknowledging the multi-perspective nature of learning phenomena, enable educational practitioners to build formal models reflecting their particular conception of learning scenarios.

In the conduct of this thesis I have sought to draw on a number of subjects from diverse domains, which obliged me to fan out considerably the number of questions investigated – perhaps more than I should have. But at all moments there was one specific variable under control and to which all the rest was somehow subjected: the designer intention. In other words, the question underlying this investigation was: *how can we ascertain that we captured the practitioner intentions*? About halfway this long period of research, it dawned on me the answer to this question that originated this dissertation: by letting them do it themselves. The rest of the investigation was then directed at researching this possibility and particularly at investigating different theories, technologies, best practices, etc. that might help me conceive an approach and, if possible, a prototype enabling domain experts in the field of education to factor out their know-how in formal representations and to build themselves their own programs.

To the question posed by (Schön, 1987) "What kinds of knowledge and what sort of know-how should teachers have in order to do their jobs well?" this work provides no answer. What it provides is a means through which, once teachers have this answer, they can specify it in the form of computer programs as expressive and nuanced as the ideas that cross their minds. That is, while the educational “epistemology of practice”, to use Schön’s own words, is to be unraveled by the practitioners themselves, this dissertation discusses a way of translating it into computer programs.

6.2 Contributions

The main contribution of this thesis is to devise an approach that empowers practitioners to create applications dovetailed to their ideas and designed to be *extensions of their experiences*. I do not argue that this approach will necessarily afford better educational tools. To claim that it will would surely be to overstate the case. I only affirm that this approach will enable the development of tools more in line with what instructors, teachers, learning designers, in sum, those who are at close range to the actual educational experiment, consider to be relevant in their everyday practices.

Furthermore, because the Multi-EML departs significantly from the existing approaches, it opens a new avenue to be explored by researchers in computer enhanced education. By allowing software to be modeled after the practitioners' own knowledge and, above all, by affording practitioners a means of reflecting on their own educational practices, I believe this approach takes up Wright Mills' challenge to meet our obligation to "provide effective guidance for the development of social practices" as well as it addresses Carroll's plea for computer applications that are accessible to non-technologists and that are able to "smoothly augment human activities".

In addition to the Multi-EML approach, this research contributes a prototype, the MDEduc. This prototype has only been feasible because of recent advances in the field of software engineering, and more notably the emergence of the Language-Oriented Programming (LOP), which many consider to be the next mainstream programming paradigm. That is, an implementation has been proposed that uses LOP principles to allow learning designers (a role that could be played by domain experts) to create multiple models and to effortlessly generate programs conformant to these models. In sum, it is an approach that, in the worst cases, entails the participation, at close range, of domain experts in the software development efforts, and that, in the best cases, leverages the idea that domain experts can develop themselves their own programs - a concept that Fowler dubbed *lay programming*.

The prototype proposes a solution that dodges the commonsense – and almost unanimous – practice of working with learning standards. It has been argued in this work that, while standardization works well for domains more limited in scope and with well defined business rules, it seems that it does more harm than good when it comes to describing complex

domains with subjective and adaptable rules - as I believe to be the case of the design of learning scenarios and of education at large.

As stated in Chapter Three, standardization favors the logic of the market, which is not necessarily in line with the needs of users for more expressive learning scenarios. Certainly, standards are valuable assets that can and should be used, if they truly match the needs of users. If this is not the case, then there is no reason to insist on the standards' road. Empowered with the ability of quickly and easily generate applications tailored to their circumstantial needs, designers do not have to subject themselves to the imperatives of the market. Thus, another outcome of this thesis that could be considered - not without a pinch of condescendence – as a contribution, is its offering a practical framework that liberates learning designers from the edicts of standards, allowing them to be systematic - because it is necessary - while leaving space for interpretation.

Another contribution of the thesis is to bring the power of the language-oriented programming (LOP), and more particularly of the Eclipse/EMF framework, to the domain of learning design. As exposed in Chapter Four, no previous work proposing the Eclipse/EMF tandem as a solution for the development of learning scenarios has been found.

After having adopted the LOP paradigm, in addition to the instrumental benefit of handing the power to develop applications down to domain experts, an important bonus was automatically obtained: a significant reduction of the life-cycle of development of learning applications. In fact, any approach that, even if technically unchallenging, required from the users a considerable amount of time to design and implement automated learning scenarios would probably be doomed to failure.

As explained in Chapter Four, LOPs implementations and particularly Eclipse/EMF allows practitioners to elaborate models and instantaneously generate applications, enabling them to promptly see their models in action, what may constitute a motivating factor. Also, with LOP's significantly reduced development cycles, I believe we can more rapidly narrow the gap between instructors' expectations and educational outcomes.

I could also add the contribution of having introduced design patterns for pedagogical approaches - or pedagogical patterns – as the starting point in the design of learning scenarios. Considering that design patterns are condensed representations of best practices from the

domains they refer to, they perfectly fulfilled the need for an informal specification with which to initiate the Multi-EML approach.

Furthermore, looking at the actual contribution offered to final users, i.e. the learning applications to be used by learners, we realize one of the most fundamental departures of the Multi-EML – and of the MDEduc prototype - from other approaches (i.e. ISD and EML). Contrary to them, we don't have a single "box" (be it a *black* box or a *transparent* box as proposed, respectively, by the ISD and EML approaches) in which our scenario model will be executed. Given that designers may come out with any conceptualization they find relevant for their specific cases and that these concepts can be mapped into virtually any existing platform description – and, in our case, into JET templates --, there is no way of telling *a priori* what the final application(s) will be. For example, it could be a web application conformant to the **peer feedback** pedagogical pattern, a graphical editor for **problem-based scenarios** or any combination of a scenario model and a platform specification.

6.3 Future work

One of the flaws of the approach here proposed is that it has not been tested among practitioners. Though the ideal outcome for a proposition of this nature would be to have final users actually working with it and providing feedbacks that evinced better results than the ones obtained with alternative approaches (e.g. EML or ISD), this is a stage I have not attained yet. In fact, validating it with real users, even if it has always been considered, would not fit the amount of work doable in this thesis. I'm convinced that this step alone would require a thesis of its own.

So, this section describes how future works could go about some of the pending issues. In particular, I envisage three main questions that have been raised - but that remain unanswered - that could direct further investigations: lowering the entry level of the design of learning scenarios, improving the quality of designed scenarios and extending the existing prototype.

6.3.1 Simplifying the design with processes

Revisiting the classification of the types of design in two categories, process-centric and entity-centric, as proposed in Chapter Two, the MDEduc prototype offers a programming environment with which learning designers can specify the entities -- i.e. a typical entity-

centric design -- of the scenarios they want to build (e.g. the concepts involved in the Peer Feedback scenario, like artifact, reviewer, etc). Later on, these entities will pass through a process-centric design to give rise to scenario instances (e.g. an actual peer feedback scenario). Thus, in an effort to reduce the scenario creation learning curve, the process of designing scenario instances and the initial activity of creating the entities could be both guided by automated wizards specifying the steps to be followed to arrive at their respective products (i.e. the scenario metamodels and the instance scenarios).

In fact, the idea of using processes to guide the creation of artifacts is not new. A particularly notable example is provided by the UML specification language, used in software engineering. In practice, the UML object oriented model is an assortment of diagrams that can be used in many different ways to attain the goal of building software programs. Many times this ultimate goal is not achieved and the program development is abandoned, due to diverse causes. The creators of the UML, diagnosing the characteristics of different failed software projects, proposed the Rational Unified Process, or RUP (Kruchten, 2004), a software development process that guides users on how to use UML diagrams in a specific project toward a goal. In sum, RUP is a process-based design that guides the use of the different parts of an underlying structure-centric design (i.e. the UML object-oriented model).

Likewise, MDEduc basically provides a structural approach to creating learning scenarios. However it does not tell how to do so, even if it suggests *en gros* a natural sequence of steps to be followed: the whole process starts with the writing of a pedagogical pattern using the PPEditor; next new metamodels will be created using the EduModel; finally the applications are generated using the EduGen. However, no help is provided to designers instructing the specific steps they should follow when conceiving both metamodels and instance models. Such assistance would lower the entry level to scenario design.

Thus, the idea is to propose finer grained processes guiding designers, just like the RUP method helps UML users. Once one such process is conceived, the Eclipse Framework offers a wide range of tools to implement them, with different entry points, aimed at reducing the learning curve: wizards, cheat sheets, templates, preference pages, property pages, help dialogs, etc. An initial investigation has already been conducted, as reported in (LePallec et al., 2006).

6.3.2 Improving the Quality of Scenarios

In the course of a design, designers have to make many decisions. Of course making all design decisions in a deliberate way is not the best approach, what takes us to some important questions underlying this work: "what can be considered a 'good' design choice?" Chapter Three listed some aspects, like expressivity, specificity, etc. that could guide designers into making some decisions. However they are too generic and cannot ascertain that final models will be "good" for specific domains, and in our case, for the design of learning scenarios.

For example, considering that the MDEduc recommends to start the creation of a learning scenario with a pedagogical pattern, how could we make sure that design decisions made really capture the *intention* of the pattern?" In this process of migrating from one representation full of semantic ambiguity to a more constrained one, we will eventually face issues about *conformance* to the source representation and the *quality* of the target one.

Sometimes we just cannot talk about conformance, since there are cases in which designers don't really want to be conformant to any pre-existing specification but rather to build their own highly personalized scenarios, in an approach that is the opposite of using standardized EMLs. At other times, however, designers do want to reuse some already established scenarios and that without having to resort to all-encompassing standards, as they have specific needs demanding specific concepts. What to do?

We have no final answer to these questions. One approach to be investigated is the open source movements. Presenting well-known examples like the Open Source (DiBona, Ockman, & Stone, 1999) and the Open Content (OCA, 2007), open source movements' rationale is to accept contributions from anybody. Then, in a process of continuous and successive improvements by the members of a given community, the initial contributions tend to acquire the desired characteristics. In our case, these contributions could be: informal learning scenario specifications (e.g. pedagogical patterns, lesson plans, etc.), formal models describing scenarios (e.g. Ecore metamodels, XML schemas, Java APIs, etc.) and platform specifications in the form of JET templates. Also, programmers could contribute with

improvements to the MDEduc plug-ins. To this respect, the MDEduc source code has been made public, in the Source Forge software repository[44].

To centralize contributions and to enable the establishment of user communities, an arsenal of auxiliary internet based tools like wikis, forums, versioning systems, etc. could be provided to hold contributions as well as to implement the "social filtering" advocated by open source movements. If necessary, other more strict controls like peer-reviewing could also be provided.

6.3.3 Extending the prototype

The most direct way of extending the prototype is by adding new templates describing the different platforms to which scenarios can be deployed. This is the most technically challenging part, because templates are actual programs that have some parameterizable parts serving to customize them for different domains. When I affirm that LOP implementations hand the control of the software development over to domain experts, I assume that these templates are already available, resting to domain experts the much simpler task of inputting the right parameters (i.e. their scenario metamodels) to the templates. Usually this simple task can be even further alleviated from learning designers, by hiding it behind amenable graphical interfaces (buttons, context menus, etc).

However, somebody still has to write the templates, a task that requires highly skilled programmers. For the MDEduc prototype -- more specifically for the EduGen plug-in -- I provided the templates for a web application (i.e. servlets, JSPs, HTML pages, deployment descriptors, DAOs, DTOs and ANT scripts) that allows basic operations with the scenario data. In addition, by default, the stock EMF comes with templates allowing to generate editors for the models. Any other envisaged platform will need its own template(s). It is by enriching the range of possible platforms that we will enable truly expressive scenarios to be elaborated, after all the semantics of scenarios reside in the programs. Thus, a considerable effort is required to develop templates to some of the most important platforms.

---

[44] http://sourceforge.net/projects/mdeduc

6.4 Final remarks

> "There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy" (William Shakespeare)

This work is just a starting point for a novel way of considering the design of learning scenarios. Instead of investing on models that try to expressively represent all scenarios – an undertaking I claim to be hardly possible – it proposes an approach (Multi-EML) that sets designers free to create truly expressive models for their specific requirements for a scenario.

This approach results from a conviction that in education there is no theory that accounts for all learning phenomena and that can advantageously replace all other theories in every situation. Instead, it views learning events as complex phenomena that can be interpreted through different sets of filters, reflecting different perspectives and even different paradigms. That is why it advocates the use of multiple abstractions to capture this multi-dimensional aspect of learning events.

As stated in Chapter Four, human beings have the natural vocation for making abstractions. So, Multi-EML capitalizes on this vocation and offers a way of leveraging these abstractions, by transforming them into executable programs. In so doing, it rescues the capacity of creation from the exclusive hands of far-off experts to hand it over to local ones, an approach that tends to dilute the "knowledge hegemony" among end users.

The Multi-EML approach proposes then to replace a single hierarchical structure, unspecific enough to encompass all possible pedagogical events, for networks of highly specific learning scenario metamodels. As (J. Coplien, 2000), advocating the use of multiple perspectives in the design of software systems, puts it:

> [Multiple paradigm design] supports abstractions that are more network-like than hierarchical, which makes them a better fit for complex domains with similarly complex structure.

Thus, Multi-EML assumes that education is one of these complex domains and proposes an alternative to fixed overarching hierarchical models.

The prototype described in this paper (MDEduc) is an attempt to show that we can enable domain experts to create new and original models of scenarios and derive code conforming to

these models without having to transform them into software engineers. It exploits the possibility of constructing full-fledged software applications unbeknownst to the idioms of programming languages. This possibility is afforded by the LOP paradigm and brings the complete development cycle within the reach of learning designers.

To revisit Carroll, a way of constructing better applications is by directly incorporating "descriptions of potential users and the uses they might make of an envisioned computer system into the design reasoning for that system". I personally cannot envisage a better way of achieving this than empowering users to do it themselves.

Finally, I cannot conceive, at least for the domain of learning scenario design, the adoption of a single standard (like IMS-LD, for example) as a long term solution. It is unimaginable that in ten years from now all learning designers will be unanimously thinking of learning scenarios uniquely as sequences of activities, or whatever other overarching metamodel they may fancy by then, just because this would be the only way of sharing their resources. I hope the community of learning designers does not misplace their collective attention for still longer into one single slant of something that is much richer and open to multiple interpretations.

Bibliography

Aaby, A. A. (2004). The Logical Foundations of Computer Science and Mathematics [Electronic Version]. Retrieved 21/03/2007 from http://www.cs.wwc.edu/~aabyan/Logic/index.html.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language : towns, buildings, construction*. New York: Oxford University Press.

Alexander, I. (2000). Introduction to Scenarios: A range of techniques for engineering better systems. Retrieved 22/03/2007, from http://easyweb.easynet.co.uk/~iany/consultancy/intro_to_scenarios.ppt

Allert, H. (2004). Coherent Social Systems for Learning: An Approach for Contextualized and Community-Centred Metadata. *Journal of Interactive Media in Education, 2004*(2).

Ambler, S. W. (1998). *Process patterns : building large-scale systems using object technology*. Cambridge, UK ; New York: Cambridge University Press.

Andrews, P. (2002). Object Lessons: The Future of OO Design [Electronic Version]. *Developer.com*. Retrieved 22/03/2007 from www.developer.com/design/article.php/1450841.

Arthorne, J., & Laffra, C. (2004). *Official Eclipse 3.0 Faqs*. Boston: Addison-Wesley.

ATL. (2007). ATL Project. 24/03/2007, from http://www.eclipse.org/m2m/atl/

Avgeriou, P. (2005, July 2005). *Architectural patterns revisited - a pattern language.* Paper presented at the 10th European Conference on Pattern Languages of Programs, Irsee, Germany.

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM, 21*(8).

Baggetun, R., Rusman, E., & Poggi, C. (2004). *Design Patterns For Collaborative Learning: From Practice To Theory And Back.* Paper presented at the World Conference on Educational Multimedia, Hypermedia and Telecommunications, Norfolk, USA.

Beck, K. (2006). *Implementation Patterns*. Boston, MA: Addison-Wesley.

Beck, K., & Andres, C. (2005). *Extreme programming explained : embrace change* (2nd ed.). Boston, MA: Addison-Wesley.

Bednar, A. K., Cunningham, D., Duffy, T. M., & Perry, J. D. (1992). Theory into practice: How do we link? In T. M. Duffy & D. H. Jonassen (Eds.), *Constructivism and the technology of instruction: a conversation* (pp. 17-34). Hillsdale: Lawrence Erlbaum Associates.

Bergin, J. (2001). Mining Pedagogical Patterns. Retrieved 23/03/2007, from http://www.pedagogicalpatterns.org/meetus/ecoop2001.html

Bergin, J., Eckstein, J., Manns, M. L., & Sharp, H. (2002). Feedback Patterns. *Pedagogical Patterns* Retrieved 22/03/2007, from http://pedagogicalpatterns.org/current/feedback.pdf

Blinco, K., Mason, J., McLean, N., & Wilson, S. (2004). Trends and Issues in E-learning Infrastructure Development. Retrieved 22/03/2007, from http://www.educationau.edu.au/papers/Altilab04-Trends-Issues.pdf

Bloom, B. S. (1956). *Taxonomy of educational objectives; the classification of educational goals* (1st ed.). New York,: Longmans, Green.
Bodker, S., & Christiansen, E. (1997). Scenarios as springboards in design of CSCW. In G. Bowker (Ed.), *Social Science, Technical Systems and Cooperative Work: Beyond the Great Divide* (pp. 217-233): LEA, Inc.
Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language user guide*. Reading Mass.: Addison-Wesley.
Botturi, L. (2003). *E2ML: Educational Environments Modeling Language.* University of Lugano, Lugano, Switzerland.
Boud, D., & Feletti, G. (1991). *The Challenge of problem based learning*. New York: St. Martin's Press.
Bradley, N. A., & Dunlop, M. D. (2005). Towards a Multidisciplinary Model of Context to Support Context-Aware Computing. *Human-Computer Interaction, 20*, 402-446.
Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., & Grose, T. J. (2003). *Eclipse modeling framework : a developer's guide*. Boston, MA: Addison-Wesley.
Buhler, P. A., Starr, C. W., & Weichhart, G. (2005). *A Composite Design Pattern for Stubless Web Service Invocation*. Paper presented at the IASTED International Conference on Software Engineering.
Burrell, G., & Morgan, G. (1979). *Sociological Paradigms and Organizational Analysis*. London: Heinemann.
Campbell, K., Schwier, R. A., & Kenny, R. F. (2005). Agency of the Instructional Designer: Moral Coherence and Transformative Social Practice. *Australasian Journal of Educational Technology, 21*(2), 242-262.
Caron, P. A. (2007). *Ingénierie dirigée par les modèles pour la construction de dispositifs pédagogiques sur des plateformes de formation.* l'Université des Sciences et technologies de Lille, Lille.
Caron, P. A., LePallec, X., & Sockeel, S. (2006). *Configuring a Web-Based Tool Through Pedagogical Scenarios.* Paper presented at the IADIS Virtual Multi Conference on Computer Science and Information Systems
Carroll, J. M. (1995). *Scenario-based design : envisioning work and technology in system development*. New York: Wiley.
Cartwright, N. (1983). *How the Laws of Physics Lie*. Oxford: Clarendon Press.
Cartwright, N. (1999). *The dappled world: a study of the boundaries of science*. Cambridge, UK: Cambridge University Press.
CDF. (2003). Curriculum Description Format (CDF). Retrieved 30/05/2004, from http://www.ariadne-eu.org/en/publications/references/a-cdf_descriptor.html
Chomsky, N. (1957). *Syntactic structures*. The Hague: Mouton & Co.
Clark, R. E. (1994). Media will never influence learning. *Educational Technology Research and Development, 42*(2), 21-29.
Conte, A., Fredj, M., Hassine, I., Giraudin, J.-P., & Rieu, D. (2002). A Tool and a Formalism to Design and Apply Patterns. *Lecture Notes in Computer Science, 2425/2002*, 135.
Cook, S. (2004). Domain Specific Modeling and Model Driven Architecture. *MDA Journal*, 1-10.
Cook, S., & Kent, S. (2004). Language Anatomy. In J. Greenfiel & K. Short (Eds.), *Software factories : assembling applications with patterns, models, frameworks, and tools*.
Coplien, J. (2000). *Multi-Paradigm Design.* Vrije Universiteit Brussel.
Coplien, J. O. (1996). *Software Patterns*. New York: SIGS.
Craik, K. (1943). *The Nature of Explanation*. Cambridge: Cambridge University Press.

Cuban, L. (2001). *Oversold and underused : computers in the classroom*. Cambridge, Mass.: Harvard University Press.

Cunningham, W. (1994). Tips For Writing Pattern Languages. Retrieved 22/03/2007, from http://www.c2.com/cgi/wiki?TipsForWritingPatternLanguages

Czarnecki, K. (2004). Interview with Krzysztof Czarnecki. Retrieved 21/03/2007, from http://www.codegeneration.net/tiki-read_article.php?articleId=64

Czarnecki, K., & Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Boston: Addison-Wesley.

Czarnecki, K., Eisenecker, U., Gluck, R., Vandervoorde, D., & Veldhuizen, T. (2000). *Generative Programming And Active Libraries* (Vol. 1766). Berlin; Heidelberg: Springer.

Dalziel, J. (2003). Implementing Learning Design: The Learning Activity Management System (LAMS). Retrieved 12/04/2007, from http://www.lamsfoundation.org/CD/html/resources/whitepapers/ASCILITE2003%20Dalzie%20Final.pdf

Dasgupta, S. (1992). *Herbert Simon's `Science of Design': two decades later*. Paper presented at the First International Conference on Intelligent Systems Engineering, Edinburgh, UK.

de Moura Filho, C. O., & Derycke, A. (2005). *Pedagogical patterns and learning design: When two worlds collide*. Paper presented at the UNFOLD-PROLEARN Joint Workshop, Valkenburg, the Nederlands.

DiBona, C., Ockman, S., & Stone, M. (1999). *Open sources : voices from the open source revolution* (1st ed.). Beijing ; Sebastopol, CA: O'Reilly.

Dillenbourg, P. (2003). *Over-scripting CSCL: The risks of blending collaborative learning with instructional design*. Paper presented at the First SCIL Congress. Retrieved 22/03/2007, from http://www.scil.ch/congress-2003/program-09-10/docs/09-track-1-1-txt-dillenbourg.pdf.

Dillon, A., & Gabbard, R. (1998). Hypermedia as an educational technology: a review of the empirical literature on learner comprehension, control and style. *Review of Educational Research, 68*(3), 322-349.

Dmitriev, S. (2004). Language Oriented Programming: The Next Programming Paradigm. Retrieved 24/03/2007, from http://www.onboard.jetbrains.com/is1/articles/04/10/lop/

Dodero, J. M., & Diez, D. (2006). *Model Driven Instructional Engineering to Generate Adaptable Learning Materials*. Paper presented at the ICALT'06, Kerkrade, The Nederlands.

Dumoulin, C. (2004). The Transformation Engine: ModTransf. Retrieved 24/03/2007, from http://www2.lifl.fr/~dumoulin/mdaTransf/doc/modTransf.doc.

E-LEN. (2005). Design expertise for e-learning centres: Design patterns and how to produce them. Retrieved 21/03/2007, from http://www2.tisip.no/E-LEN/documents/ELEN-Deliverables/booklet-e-len_design_experience.pdf

Eclipse. (2007). A presentation of the technical aspects of the Eclipse Project. from http://www.eclipse.org/eclipse/presentation/eclipse-slides.html

ECOM. (2000, 20/03/2007). Activity Report. Retrieved 21/03/2007, from http://www.ecom.jp/ecom_e/latest/ecomjournal_no2/report_e/wg_mec1_mec2_e02.htm

EFX. (2007). The EFx Architectural-Guidance Software Factory. Retrieved 25/03/2007, from http://msdn2.microsoft.com/en-us/library/aa905331.aspx

El-Kechaï, H., & Choquet, C. (2005). *Approche pragmatique de conception d'un EIAH : réingénierie pédagogique dirigée par les modèles.* Paper presented at the Environnements Informatiques pour l'Apprentissage Humain, Montpellier.
Elder, P. (2005). JET Enhancement Proposal (JET2). Retrieved 24/03/2007, from http://www.eclipse.org/modeling/emf/docs/architecture/jet2/jet2.html
EML. (2000). Educational Modelling Language. Retrieved 22/03/2007, from http://hdl.handle.net/1820/81
Engeström, Y., Miettinen, R., & Punamäki-Gitai, R.-L. (1999). *Perspectives on activity theory*. Cambridge ; New York: Cambridge University Press.
Fabos, B., & Young, M. D. (1999). Telecommunications in the classroom: Rhetoric versus reality. *Review of Educational Research, 69*(3), 217-259.
Farley, F. H. (1982). The future of educational research. *Educational Researcher, 11*(8), 11-19.
Ferraris, C., Lejeune, A., Vignollet, L., & David, J. P. (2005). *Modélisation de scénarios d'apprentissage collaboratifs pour la classe.* Paper presented at the EIAH 2005, Montpellier, France.
Feyerabend, P. (1975). *Against method : outline of an anarchistic theory of knowledge*. Atlantic Highlands, N.J.: Humanities Press.
Fields, R. E. (2001). *Analysis of erroneous actions in the design of critical systems.* University of York, York, England.
Figueiredo, A. D., & Afonso, A. P. (2005). Context and Learning: A Philosophical Framework. In A. D. Figueiredo & A. P. Afonso (Eds.), *Managing Learning in Virtual settings: The Role of Context*. Hershey, PA: Information Science Publishing (Idea Group).
Floyd, C. (1987). Outline of a paradigm change in software engineering. In G. Bjerknes, P. Ehn, M. Kyng & K. Nygaard (Eds.), *Computers and democracy : a Scandinavian challenge*. Brookfield, VT: Gower.
Foster, I., & Kesselman, C. (1998). *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA: Morgan Kaufmann Publishers Inc.
Fowler, M. (1997). *Analysis patterns : reusable object models*. Menlo Park, Calif.: Addison Wesley.
Fowler, M. (2004). *UML distilled : a brief guide to the standard object modeling language* (3rd ed.). Boston: Addison-Wesley.
Fowler, M. (2005). Language Workbenches [Electronic Version]. Retrieved 21/03/2007 from http://www.martinfowler.com/articles/languageWorkbench.html.
Fowler, M. (2005). Language Workbenches and Model Driven Architecture [Electronic Version]. Retrieved 21/03/2007 from http://www.martinfowler.com/articles/mdaLanguageWorkbench.html.
Frankel, D. S. (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing*. Indianapolis, IN: John Wiley & Sons.
Friesen, N. (2004). The E-learning Standardization Landscape. Retrieved 22/03/2007, from http://www.cancore.ca/docs/intro_e-learning_standardization.html
Frigg, R., & Hartmann, S. (2006). Models in Science [Electronic Version]. *The Stanford Encyclopedia of Philosophy*. Retrieved 21/03/2007 from http://plato.stanford.edu/archives/spr2006/entries/models-science/.
Gaeta, A., Ritrovato, P., & Gaeta, M. (2005). *Towards a Domain Specific Application Development Environment for the ELeGI architecture: the Software Factories approach*. Paper presented at the First International ELeGI Conference on Advanced Technology for Enhanced Learning.

Gagne, R. (1985). *The Conditions of Learning* (4th ed.). New York: Holt, Rinehart & Winston.
Gamma, E., & Beck, K. (2003). *Contributing to Eclipse: principles, patterns, and plugins*. Reading, Mass: Addison-Wesley.
Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1995). *Design patterns : elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley.
Gentner, D., & Stevens, A. L. (1983). *Mental Models*. Hillsdale, NJ: Lawrence Erlbaum Associates.
Gerber, A., & Raymond, K. (2003). *MOF to EMF: there and back again*. Paper presented at the Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange.
Gibbons, A. S., & Rogers, P. C. (2006a). *Coming At Design From A Different Angle: Functional Design*. Paper presented at the Association for Educational Communications and Technology.
Gibbons, A. S., & Rogers, P. C. (2006b). A Selection of Quotations on Design Languages and Their Relation to Design Layers [Electronic Version]. Retrieved 21/03/2007 from http://it.coe.uga.edu/itforum/paper94/Paper94.pdf.
Gibbons, A. S., & Rogers, P. C. (2007). The Architecture of Instructional Theory. In C. M. R. A. Carr-Chellman (Ed.), *Instructional-Design Theories and Models* (Vol. III). Mahwah, NJ: Lawrence Erlbaum Associates.
Goodyear, P. (2005). Educational design and networked learning: Patterns, pattern languages and design practice. *Australasian Journal of Educational Technology, 21*(1), 82-101.
Grafinger, D. J. (1988). Basics of instructional systems development. In *INFO-LINE* (Vol. 8803). Alexandria: American Society for Training and Development.
Greenfield, J., & Short, K. (2004). *Software factories : assembling applications with patterns, models, frameworks, and tools*. Indianapolis, IN: Wiley Pub.
Griffiths, D., Blat, J., Garcia, R., Vogten, H., & Kwong, K. L. (2005). Learning Design Tools. In R. Koper & C. Tattersall (Eds.), *Learning design : a handbook on modelling and delivering networked education and training*. Berlin: Springer.
Grob, H. L., Bernsberg, F., & Dewanto, B. L. (2005). Model Driven Architecture (MDA): Integration and Model Reuse for Open Source eLearning Platforms [Electronic Version]. *eleed*. Retrieved 26/03/2007 from http://eleed.campussource.de/archive/1/81/.
Gruber, T. R. (1995). Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *Int. Journal of Human-Computer Studies, 43*, 907-928.
Guba, E., & Lincoln, Y. (1994). Competing paradigms in qualitative research. In N. Denzin & Y. Lincoln (Eds.), *Handbook of Qualitative Research*. Thousand Oaks, CA: Sage Publications.
Gupta, G. (2002). *A Language-centric Approach to Software Engineering: Domain Specific Languages meet Software Components.* Paper presented at the CologNet Workshop on Logic Programming, Madrid.
Hannafin, M. J., Hannafin, K. M., Land, S. M., & Oliver, K. (1997). Grounded practice and the design of constructivist learning environments. *Educational Technology Research and Development, 45*(3), 101-117.
Harel, D., & Marelly, R. (2003). *Come, let's play : scenario-based programming using LSCs and the play-engine*. Berlin ; New York: Springer.
Haywood, D. (2004). MDA: Nice Idea. Shame about the... [Electronic Version]. Retrieved 21/03/2007 from http://www.theserverside.com/tt/articles/article.tss?l=MDA_Haywood.

Heering, J., & Mernik, M. (2002). *Domain-specific languages for software engineering*. Paper presented at the 35th Annual Hawaii International Conference on System Sciences, Hawaii.
Hollingsworth, C. D. (2005). *Martin Heidegger's Phenomenology And The Science Of Mind.* Louisiana State University.
Howe, C. Z. (2006). Considerations when using phenomenology in leisure inquiry: Beliefs, methods, and analysis in naturalistic research. *Leisure Studies, 10*(1), 49-62.
IDL. (2002). Specification of OMG Interface description language. Retrieved 25/03/2007, from http://www.omg.org/cgi-bin/doc?formal/02-06-39
IMS-EP. (2005). ePortfolio: Best Practice Guide, Binding, Information Model, Rubric Specification, XML Examples and Schemas Version 1 Final Specification. Retrieved 22/03/2007, from http://www.imsglobal.org/ep/index.html
IMS-LD. (2003). Learning Design: Information Model, Best Practice and Implementation Guide, XML Binding, Schemas. Version 1.0 Final Specification. Retrieved 22/03/2007, from http://imsglobal.org/learningdesign/
IMS-LI. (2005). Learner Information Package: Information Model, Package XML Binding Specification, Best Practices and Implementation Guide. Version 1.0.1 Final Specification. Retrieved 22/03/2007, from http://www.imsglobal.org/profiles/index.html
IMS. (2006). IMS Global Learning Consortium. Retrieved 28/11/2006, from http://imsglobal.org/
ISO. (2002). Information Technology: Learning by IT. *ISO Bulletin* Retrieved 30/03/2007, from http://jtc1sc36.org/doc/36N0264.pdf
ISO. (2005, 04/07/2005). ISO - International Organization for Standards - FAQ - Standards. Retrieved 23/03/2007, from http://www.iso.org/iso/en/faqs/faq-standards.html
Jackson, M. (2001). *Problem frames: analyzing and structuring software development problems*. Harlow, England: Addison-Wesley Longman Publishing Co., Inc.
JISC. (2004). Designing for Learning: An update on the Pedagogy strand of the JISC eLearning Programme. Retrieved 21/03/2007, from http://www.jisc.ac.uk/uploaded_documents/Overview.doc
Johnson-Laird, P., & Byrne, R. (2000). Mental Models Website. Retrieved 29/03/2007, from http://www.tcd.ie/Psychology/Ruth_Byrne/mental_models/
Jonassen, D. H. (1994). Toward a Constructivist Design Model. *EDUCATIONAL TECHNOLOGY, 34*(4), 34.
Kermeta. (2007). Kermeta Project. Retrieved 24/03/2007, from http://www.kermeta.org/
Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA explained: the model driven architecture: practice and promise*. Boston: Addison-Wesley.
Knuth, D. E. (1992). *Literate Programming*. Stanford, CA: Center for the Study of Language and Information.
Knuth, D. E., & Levy, S. (2002). *CWEB User Manual (VERSION 3.64)*. Retrieved 21/03/2007, from http://www.literateprogramming.com/cweb.pdf.
Koper, R. (2002). Modeling units of study from a pedagogical perspective – The pedagogical metamodel behind EML Retrieved 22/03/2007, from http://hdl.handle.net/1820/36
Koper, R., & Tattersall, C. (2005). *Learning design : a handbook on modelling and delivering networked education and training*. Berlin ; New York: Springer.
Krishnamurthi, S. (2003). *Programming Languages: Application and Interpretation*. Retrieved 23/03/2007, from http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/.

Kruchten, P. (2004). *The rational unified process : an introduction* (3rd ed.). Boston: Addison-Wesley.
Kuhn, T. S. (1962). *The structure of scientific revolutions*. Chicago: University of Chicago Press.
Kyng, M. (1995). Creating Contexts for Design. In J. M. Carroll (Ed.), *Scenario Based Design: Envisioning Work and Technology in System Development*. New York: Wiley.
Laforcade, P. (2004). *Méta-modélisation UML pour la conception et la mise en oeuvre de situations-problèmes coopératives.* L'Université de Pau et des Pays de l'Adour.
Laforcade, P. (2007). *Scénarisation Pédagogique et Ingénierie Dirigée par les Modèles - Cadre d'étude pour la définition de langages et environnement - outils de scénarisation pédagogique spécifiques à des domaines.* Paper presented at the EIAH'07, Lausanne, Switzerland.
LePallec, X. (2002). *Des services d'adaptation de modèles pour la coopération de méta-systèmes : application aux groupware flexibles.* Université de Lille 1, Lille.
LePallec, X., De Moura Filho, C. O., Marvie, R., Nebut, M., & Tarby, J. C. (2006). *Supporting generic methodologies to assist IMS-LD modeling.* Paper presented at the Sixth IEEE International Conference on Advanced Learning Technologies, Kerkrade, The Netherlands.
Liddle, D. (1996). Design of the Conceptual Model: An interview with David Liddle. In T. Winograd (Ed.), *Bringing Design to Software*. Reading, Mass: Addison-Wesley.
LOM. (2002). *Standard for Learning Object Metadata* (No. 148.41.21): Learning Technologies Standards Committee of the IEEE.
Lyotard, J. F. (1979). *La condition postmoderne : rapport sur le savoir*. Paris: Éditions de Minuit.
MacLeod, C., & Northover, S. (2001). SWT: The Standard Widget Toolkit - PART 2: Managing Operating System Resources. Retrieved 25/03/2007, from http://www.eclipse.org/articles/swt-design-2/swt-design-2.html
Martel, C., Vignollet, L., Ferraris, C., David, J.-P., & Lejeune, A. (2006). *LDL: an alternative EML.* Paper presented at the Sixth International Conference on Advanced Learning Technologies, Kerkrade, The Nederlands.
Martens, H., & Vogten, H. (2005). A Reference Implementation of Leaning Design Engine. In R. Koper & C. Tattersall (Eds.), *Learning Design: a handbook on modelling and delivering networked education and training*. Berlin ; New York: Springer.
Martin, R., Riehle, D., & Buschmann, F. (1998). *Pattern Languages of Program Design 3*. Reading, MA.: Addison-Wesley.
McMullin, E. (1968). What Do Physical Models Tell Us? In B. van Rootselaar & J. F. Staal (Eds.), *Logic, Methodology and Science III* (Vol. III, pp. 385-396). Amsterdam: North Holland.
MDA. (2003). *MDA Guide Version 1.0.1*: OMG.
MDA_Wikipedia. (2007). Model Driven Architecture. Retrieved 24/03/2007, from http://en.wikipedia.org/w/index.php?title=Model-driven_architecture&oldid=114279465
Megginson, D. (2005). *Imperfect XML : rants, raves, tips, and tricks-- from an insider*. Upper Saddle River, NJ: Addison Wesley.
Meirieu, P. (1987). *Apprendre…oui mais comment*. Paris: ESF.
Merks, E., Steinberg, D., Hussey, K., & Damus, C. W. (2007). Effective use of the Eclipse Modeling Framework. Retrieved 25/03/2007, from http://www.eclipsecon.org/2007/index.php?page=sub/&id=3619

ModFact. (2003). The ModFact Open Source project. Retrieved 24/03/2007, from http://modfact.lip6.fr/
MOF. (2002). Meta-Object Facility (MOF) v1.3.1. Retrieved 24/03/2007, from http://www.omg.org/technology/documents/formal/mof.htm
Molenda, M. (2003). In search of the elusive ADDIE model. *Performance Improvement, 42*(5), 34-36.
MPPT. (2007). Microsoft Patterns and Practices Developer Center. Retrieved 25/03/2007, from http://msdn.microsoft.com/practices/
Nardi, B. A. (1995). Some Reflections on Scenarios. In J. M. Carroll (Ed.), *Scenario Based Design: Envisioning Work and Technology in System Development*. New York: Wiley.
Nardi, B. A. (1996). Activity Theory and Human-Computer Interaction. In B. A. Nardi (Ed.), *Context and Consciousness: Activity Theory and Human-Computer Interaction* (pp. 7-16). Cambridge, Massachusetts: MIT Press.
Nodenot, T. (2005). *Contribution à l'ingénierie dirigée par les modèles en EIAH: le cas des situations-problèmes coopératives.* Université de Pau et des Pays de l'Adour, Pau, France.
Nodenot, T., & Laforcade, P. (2006). *CPM: A UML Profile to Design Cooperative PBL Situations at Didactical Level* Paper presented at the Sixth International Conference on Advanced Learning Technologies, Kerkrade, The Netherlands.
Northover, S. (2001). SWT: The Standard Widget Toolkit (PART 1: Implementation Strategy for Java™ Natives). Retrieved 25/03/2007, from http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html
Oakhill, J., & Garnham, A. (1996). *Mental Models in Cognitive Science*. Mahwah, NJ: Lawrence Erlbaum Associates.
OCA. (2007). *Open Content Alliance*. Retrieved 08/04/2007, from http://www.opencontentalliance.org/
Olivier, B., & Tattersall, C. (2005). The Learning Design Specification. In R. Koper & C. Tattersall (Eds.), *Learning design : a handbook on modelling and delivering networked education and training*. New York; Berlin: Springer.
OMG. (1997). UML Semantics. Retrieved 23/03/2007, from http://www.omg.org/docs/ad/97-07-07.ps
OUNL. (2007). OUNL Mission. Retrieved 23/03/2007, from http://www.ou.nl/eCache/DEF/71/502.html
Paquette, G. (2004). Educational Modeling Languages: From an Instructional Engineering Perspective. In R. McGreal (Ed.), *Online education using learning objects* (pp. 331-346). London: Routledge/Palmer.
Parets-Llorca, J., & Grunbacher, P. (1999). *Capturing, negotiating, and evolving system requirements: bridgingWinWin and the UML.* Paper presented at the 25th Euromicro Conference: Proceedings : Informatics : Theory and Practice for the New Millennium Milan, Italy.
Pernin, J. P., & Lejeune, A. (2004). *Modèles pour la réutilisation de scénarios d'apprentissage.* Paper presented at the TICE Méditerranée, Nice.
Popma, R. (2004a). JET Tutorial Part 1 (Introduction to JET) [Electronic Version]. Retrieved 21/03/2007 from http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html.
Popma, R. (2004b). JET Tutorial Part 2 (Write code that writes code) [Electronic Version]. Retrieved 21/03/2007 from http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html.
QVT. (2005). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Retrieved 24/03/2007, from http://www.omg.org/docs/ptc/05-11-01.pdf

Rawlings, A., Rosmalen, P., Koper, R., Rodriguez-artacho, M., & Lefrere, P. (2002). *Survey of Educational Modelling Languages (EMLs)*. Retrieved 22/03/2007. from http://www.cen.eu/cenorm/businessdomains/businessdomains/isss/activity/emlsurveyv1.pdf.

Reigeluth, C. M. (1983). *Instructional-design theories and models*. Hillsdale, N.J.: Lawrence Erlbaum Associates.

Renaux, E., Caron, P. A., & Le Pallec, X. (2005). *Learning Management System component based design: a model driven approach.* Paper presented at the MCETECH'2005, Montreal.

Rheinfrank, J., & Evenson, S. (1996). Design Languages. In T. Winograd (Ed.), *Bringing Design to Software*. New York, NY: Addison-Wesley.

Rieu, D., Giraudin, J.-P., Saint-Marcel, C., & Front-Conte, A. (1999). *Des opérations et des relations pour les patrons de conception.* Paper presented at the XVII Congrès INFORSID, La Garde, France.

Robinson, M. (1997). "As real as it gets..." Taming Models and Reconstructing Procedures. In *Social science, technical systems, and cooperative work : beyond the great divide* (pp. xxiii, 470 p.). Mahwah, N.J.: Lawrence Erlbaum Associates.

Rolland, C. (1993). *Modeling the Requirements Engineering Process*. Paper presented at the 3rd European-Japanese Seminar on Information Modelling and Knowledge Bases.

Rosemberg, S. (2007). Anything You Can Do, I Can Do Meta (Part II) [Electronic Version]. Retrieved 21/03/2007 from http://www.technologyreview.com/Infotech/18021/.

Scarpino, M. (2005). *SWT/JFace in action*. Greenwich, CT: Manning.

Schneider, D. K. (2004). Learning together through collaborative portal sites. In M. Tokoro & L. Steels (Eds.), *Learning Zone of One's Own: Sharing Representations and Flow in Collaborative Learning Environments*. Amsterdam: OS Press.

Schön, D. A. (1987). *Educating the reflective practitioner : toward a new design for teaching and learning in the professions* (1st ed.). San Francisco: Jossey-Bass.

Seidewitz, E. (2003). What models mean. *IEEE Software, 21*(5), 26-32.

Simon, H. A. (1996). *The sciences of the artificial* (3rd ed.). Cambridge, Mass.: MIT Press.

Simonyi, C. (1996). Intentional programming: Innovation in the legacy age. Retrieved 21/03/2007, from http://citeseer.ist.psu.edu/simonyi96intentional.html

Simonyi, C., Christerson, & Clifford. (2006). Intentional Software [Electronic Version]. Retrieved 21/03/2007 from http://www.intentsoft.com/technology/IS_OOPSLA_2006_paper.pdf.

Sommerville, I. (1989). *Software engineering* (3rd ed.). Wokingham, England ; Reading, Mass.: Addison-Wesley.

Stepper, E. (2006). Sympedia GenFw Tutorial 2: Generating Extension Points. Retrieved 21/07/2007, from http://genfw.berlios.de/tutorial2/SympediaGenFwTutorial2.html

Supekar, K., Patel, C., & Lee, Y. (2002). *Characterizing quality of knowledge on semantic web.* Paper presented at the 7th International Florida Artificial Intelligence Research Society Conference, Miami, Florida.

Süß, C., & Freitag, B. (2002). *LMML - The Learning Material Markup Language Framework.* Paper presented at the International Worshop ICL, Villach, Austria.

Tattersall, C., Burgos, D., & Koper, R. (2006). An Open eLearning Specification for Multiple Learners and Flexible Pedagogies. Retrieved 23/03/2007, from http://hdl.handle.net/1820/578

Thomas, D. (2003). UML - Unified or Universal Modeling Language? *Journal of Object Technology, 2*(1), 7-12.

Tristram, C. (2003). Everyone's a Programmer [Electronic Version]. *Technology Review*. Retrieved 21/03/2007 from http://www.technologyreview.com/Infotech/13377/.

UML. (2004). Unified Modeling Language Specification. version 2.0. Retrieved 25/03/2007, from http://www.omg.org/cgi-bin/doc?ptc/2004-10-05

Vachet, C. (2006). *Scénarisation Pédagogique, Situation Interactionnelle et IDM*. Paper presented at the IDM'06. Retrieved 26/03/2007, from http://www.syscom.univ-savoie.fr/publi/Vachet06b.pdf.

van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices, 35*(6), 26-36.

van Diggelen, W., Overdijk, M., & De-Groot, R. (2005). *'Say it out loud in writing': A dialectical inquiry into the potentials and pitfalls of computer supported argumentative discussions.* Paper presented at the CSCL 2005, Taipei, Taiwan.

van Emde Boas, G. (2004). *Template Programming for Model-Driven Code Generation.* Paper presented at the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, CA.

van Es, R., & Koper, R. (2006). Testing the pedagogical expressiveness of IMS LD. *Educational Technology & Society, 9*(1), 229-249.

Verplank, Fulton, Black, & Moggridge. (1993). Scenario Guidelines. Retrieved 22/03/2007, from http://research.dh.umu.se/niklas/cameo/scen_deep.html

Visscher-Voerman, I., & Gustafson, K. L. (2004). Paradigms in the theory and practice of education and training design. *Educational Technology Research and Development, 52*(2), 69-89.

Visscher-Voerman, J. I. A. (1999). *Design approaches in training and education: a reconstructive study*. Enschede, NL: PrintPartners Ipskamp.

Voelter, M. (2000). *Pattern Refactoring*. Paper presented at the OOPSLA 2000. Retrieved 22/03/2007, from http://www.laputan.org/patterns/positions/Voelter.pdf.

Wang, H., & Zhang, D. (2003). *MDA-based Development of E-Learning System.* Paper presented at the 27th Annual International Computer Software and Applications Conference, New York.

Ward, M. (1994). Language Oriented Programming. *Software, Concepts and Tools, 15*, 147-161.

Wartofsky, M. W. (1979). *Models: Representation and the Scientific Understanding*. Dordrecht: D. Reidel Publishing Company.

Welie, M., Gerrit, C., & Eliens, A. (2000). *Patterns as Tools for User Interface Design*. Paper presented at the Workshop on Tools for Working With Guidelines.

Williams, R. (2000). *funnelweb tutorial manual*. Retrieved 25/03/2007, from http://www.literateprogramming.com/fwtut.pdf.

Wilson, B. (1997). The postmodern paradigm. In C. Dills & A. Romiszowski (Eds.), *Instructional development paradigms*. Englewood Cliffs, NJ: Educational Technology Publications.

Winograd, T. (1996). *Bringing design to software*. Reading, Mass: Addison-Wesley.

Winograd, T., & Flores, F. (1986). *Understanding computers and cognition : a new foundation for design*. Norwood, N.J.: Ablex Pub. Corp.

XMI. (2005). OMG's MetaObject Facility. Retrieved 24/07/2003, from http://www.omg.org/technology/documents/formal/xmi.htm

Zemsky, R., & Massy, W. F. (2003). Thwarted innovation: What happened to e-learning and why. Final Report for The Weatherstation Project of The Learning Alliance at the University of Pennsylvania. [Electronic Version]. Retrieved 31/003/2007 from http://www.irhe.upenn.edu/Docs/Jun2004/ThwartedInnovation.pdf.

Appendice

Appendix A: The Eclipse Platform

A.1 Introduction

Eclipse is a sophisticated framework and even whole books have only enough space to provide either a fair explanation of the complete framework or an advanced view focusing on any given aspect. Having not enough space in this thesis to describe it, I will provide a bird's-eye view to this framework, showing only its nuts and bolts, though for a good understanding of the MDEduc prototype (and of the Chapter Five), a much deeper explanation would be necessary.

A.2 The Eclipse Project

When we use the word Eclipse, in addition to the Eclipse *Platform*, we may be referring to the Eclipse *Project*, where the platform is developed, or even to the Eclipse *Foundation*, which maintains the project. Certainly, most of the time, unless otherwise stated, I will be referring to the platform. However, to make this point clearer, I will discuss, in this section, the Eclipse Project, pointing out where exactly in this gigantic enterprise the present work situates.

The **Eclipse Project** is constituted by a community of users, constantly extending the covered application areas. Originally, it was maintained by IBM, but now is managed by the **Eclipse Foundation**, an independent not-for-profit consortium of software industry vendors, including software tool vendors like Adobe, Borland, SAP AG, BEA Systems, IBM Rational, RedHat, OMG, HP, etc.

The software produced by the Eclipse Project is made available under the Common Public License (CPL), one of the least constraining open licenses, allowing developers to use, modify, redistribute it for free, and even include it as part of a proprietary product. On the other hand, any software contributed to the Eclipse Project must also be licensed under the CPL terms.

A.2.1 The Top Level Projects

All work at the Eclipse Project is organized under top-level projects, including the recently created Eclipse Modeling Project (EMP) - covering most areas of the Language Oriented Programming paradigm - to which the EMF belongs. Top level projects may be further sub-divided into sub-projects and components. The Eclipse top level projects are represented in Figure 53.

Figure 53: Eclipse Top Level Projects

One of the top level projects, confusingly enough, is called the Eclipse Top-Level Project, though sometimes it is referred to by "Platform Project" or "Eclipse Project", which is even more confusing[45]. The Eclipse Top-Level Project is the project responsible for building and maintaining an industry platform for the development of tools and rich client applications and is divided into four subprojects: the Platform, the Java Development Tools (JDT), the Equinox Project and the Plug-in Development Environment (PDE). Collectively, these subprojects provide everything needed to extend the framework and to develop tools based on Eclipse. We can see the Eclipse Top Level Project (or Eclipse Project) and its constituent parts below (please see Figure 54):

[45] In fact the Eclipse Top Level Project is composed of four different parts, one of which is responsible for the platform; on the other hand, the Eclipse Project comprises other top level projects, other than the Top Level Project. So, if using "Platform Project" is reducing, employing "Eclipse Project" is inappropriate.

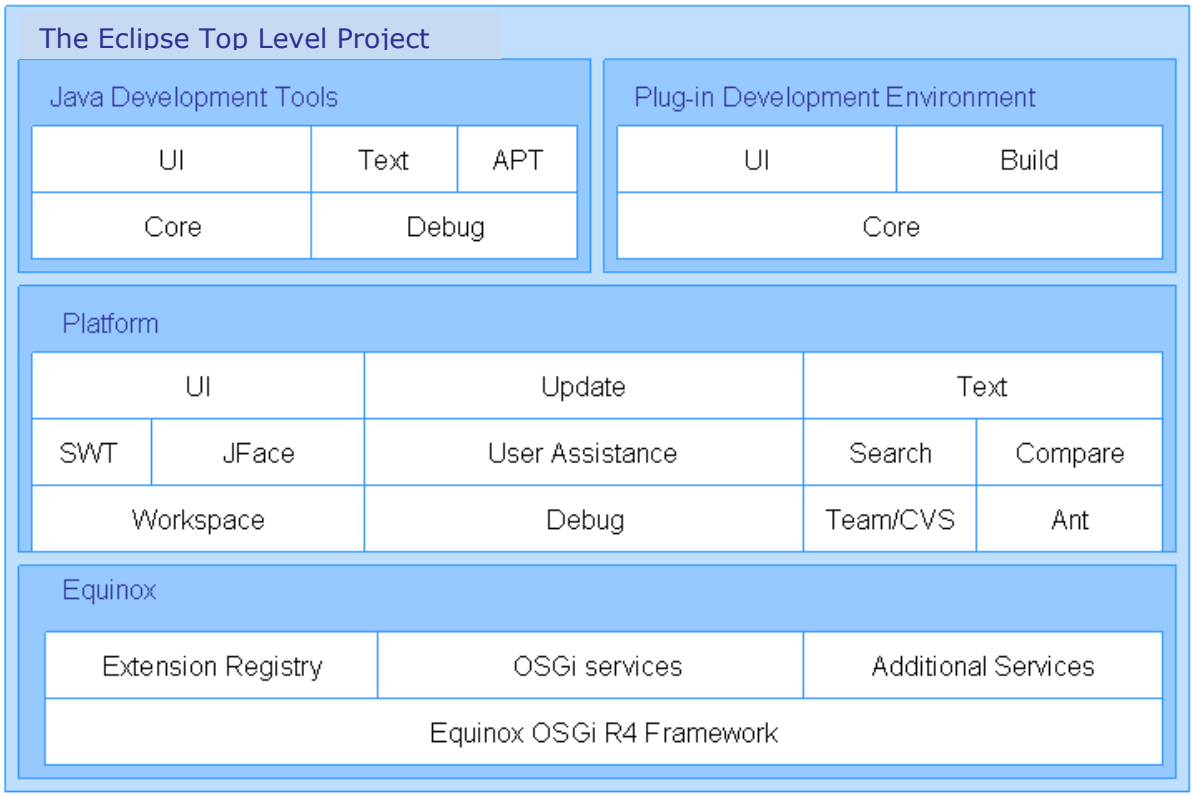


Figure 54: the Eclipse Top Level Project

Some examples of other top level projects are the Eclipse Tools Project, which serves as a "focal point" for diverse tool projects on the Eclipse Platform, thus avoiding duplicate efforts; the WTP (Web Tools Platform), which develops a platform and a set of tools for J2EE and Web-centric application development; the Test & Performance Tools Platform (TPTP) Project, which provides frameworks and services allowing developers to build test and performance tools; the Eclipse SOA Tools Platform Project (STP), dedicated to providing a platform for producing Service Oriented Architecture (SOA) applications, etc.

However, for the scope of this thesis, I made only use of the Top Level Project and of the **Eclipse Modeling Project**[46] – to be briefly presented in the next section. This is indicated in Figure 55. On the other hand, other Eclipse-based modeling applications have been used that do not belong to the Eclipse Project, like Acceleo[37], Sympedia GenFw Generator Framework[34] or Merlin Generator[38].

[46] http://www.eclipse.org/modeling/

Figure 55: Eclipse Top Level Projects used in this thesis

A.2.2 The Eclipse Modeling Project

The Eclipse Modeling Project (EMP) focuses on model-based technologies by providing a unified set of modeling frameworks, tooling, and standards implementations. For the sake of organization, EMP's projects are presently grouped into six different areas, shown in Figure 56 and explained in the sequence.

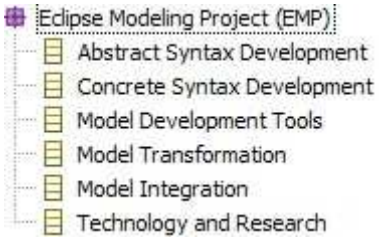


Figure 56: Eclipse Modeling Project

- **Abstract Syntax Development**: includes the Eclipse Modeling Framework (EMF), the mainstay of the whole project, which consists of a modeling framework and code generation facility for building tools and other applications based on a structured data model. It also comprises metamodels, built on the top of the EMF, that provide facilities like transaction handling (Model Transaction), model validation (Validation Framework), and many others.
- **Concrete Syntax Development**: includes a project creating an infrastructure for developing graphical editors (Graphical Modeling Framework) based on EMF.

- **Model Development Tools**: brings other industry models, like ontologies (EODM), Unified Modeling Language 2.x (UML2), XML Schema, etc. to the EMF world, allowing to-ing and fro-ing between EMF and these different technologies.
- **Model Transformation**: formed by projects providing model-to-model and model-to-text transformations. Includes, among others, the Model to Model Transformation (M2M), the Atlas Transformation Language (ATL), and the important Java Emitter Templates (JET) and JET Editor projects, which have been extensively used in the prototype described in the next chapter.
- **Technology and Research**: in this subproject, we can find many research-oriented projects contributed by institutions from various countries, including France.

Some of the main projects encompassed by the Eclipse Modeling Project are represented graphically in Figure 57.

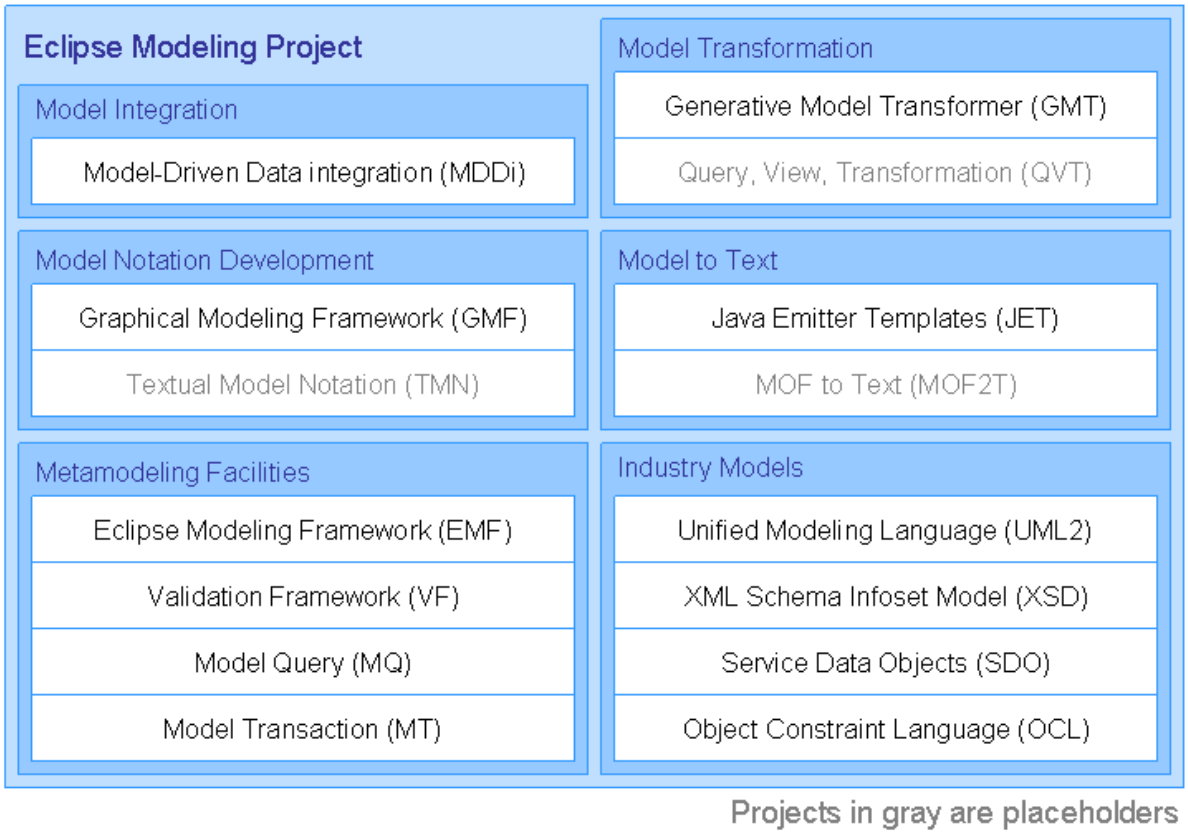


Figure 57: the Eclipse Modeling Project (source: Eclipse homepage)

In the coming section I will present the Eclipse Platform, emphasizing some aspects that have been explored in the development of the prototype

A.3 The Eclipse Platform

> To draw a computing analogy, [Eclipse] is like the Internet Protocol: exceedingly generic, not terribly interesting by itself, but a solid foundation on which very interesting applications can be built.
>
> FAQ: What is the Eclipse Platform?

Eclipse is a free software and open source platform-independent software framework for delivering "rich-client applications"[47]. Not few would describe the Eclipse platform as an IDE for programming Java, while it is a much more generic framework for constructing any type of rich client – including the Java IDE widely adopted by Java programmers all over the world. The fact that Eclipse's Java IDE – i.e., the visible part of the Java Development Tools (JDT) project - is by far the best known of such rich clients may have contributed to this misconception.

In Eclipse, "everything is a contribution" (E. Gamma & Beck, 2003), what in the Eclipse parlance could be rephrased as "everything is a **plug-in**". Apart from a very skinny kernel -- an implementation of the OSGi specification[48] known as the Platform Runtime and whose main role is to manage the plug-ins -- all of the Eclipse Platform's functionality is located in the plug-ins themselves, whether contributed by the Eclipse development team or by other members of the eclipse community.

Figure 58 shows how Eclipse's plug-in architecture differs from other plug-in extensible applications, like, for example, Firefox or Acrobat. That is, such extensible applications are functionally complete programs, but that can be extended via plug-ins directly attached to them. Conversely, the Eclipse kernel "application", in itself, does nothing in special other than

---

[47] "Rich clients support a high-quality end-user experience for a particular domain by providing rich native user interfaces (UIs) as well as high-speed local processing. Rich UIs support native desktop metaphors such as drag-and-drop, system clipboard, navigation, and customization. […] The term rich client was used to differentiate such clients from terminal client applications, or simple clients, which they replaced" (McAffer, 2005). Nowadays, however, we use this expression to contrast these applications with *browser-based* ones.

[48] http://www.osgi.org/

accepting and managing plug-ins; in that case, as already stated, it is up to the plug-ins to provide all functionality required by users. Also, with Eclipse, plug-ins do not have to connect directly to the kernel, since they can also attach themselves to other plug-ins.

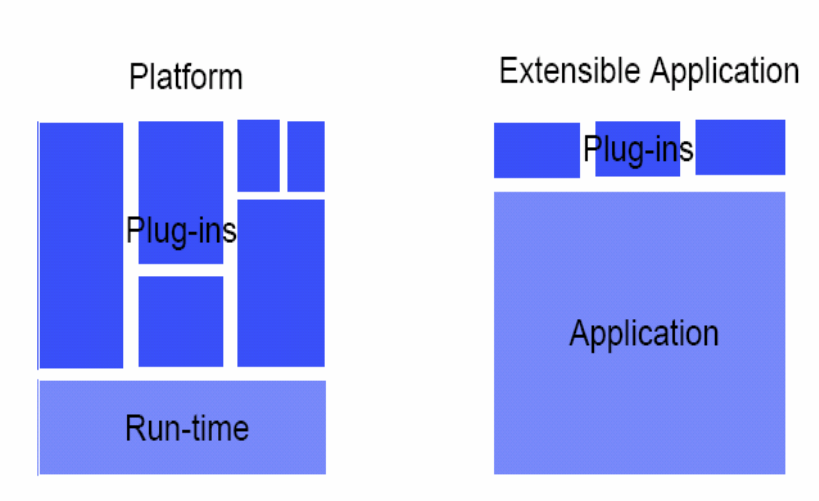


Figure 58: Eclipse plug-ins (left) compared to other extensible applications (right)

It is possible to create an application detached from the Eclipse Platform. However, building an application on the top of Eclipse Platform means that it can easily be integrated to other applications that are already installed on the Platform.

A.3.1 Inside the Eclipse platform

The Eclipse platform provides some basic level of services (please see Figure 59).

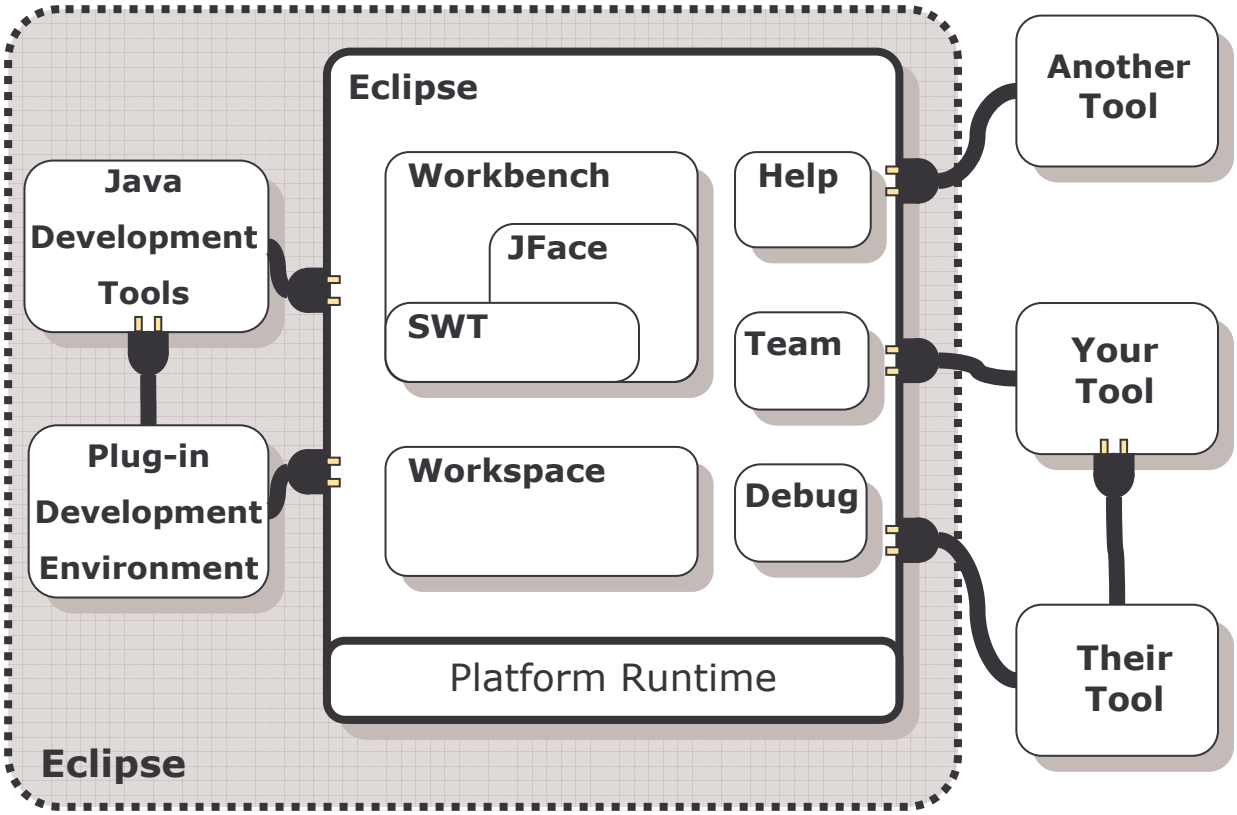


Figure 59: basic services offered by the Eclipse platform. (Source: Eclipse, 2007)

Plug-in management: The platform loads the plug-ins and manages a plug-in registry. Although a plug-in includes all it needs to run (like binary code, images, etc), part of the needed resources are present declaratively only, in two manifest files: the MANIFEST.MF, which basically lists the plug-ins to which this one depends and the *plugin.xml*, where more detailed information on the kind of dependency is listed, like, for example:

- Extension Points— declarations of functionality that it makes available to other plug-ins.
- Extensions— declarations of use of other plug-ins' extension points (i.e. a plug-in implements the functionalities required to extend other plug-ins)

These two items have not been listed by chance, but because of their importance to the whole Eclipse framework. In fact, the extension point mechanism is the way of explicitly declaring where a plug-in can be extended. As (E. Gamma & Beck, 2003) put it:

> Object-oriented frameworks are designed, as is Eclipse, to be used by being extended. One problem with object-based frameworks is that every public method is a potential point for extension. If you're successful and people use your framework, you quickly find that you have no freedom to grow the framework further. Every method has potentially been extended, so you can't

> change anything. As Martin Fowler says, we should distinguish between public and published methods. Published methods are intended for use by others, public methods are merely visible.

Thus, Eclipse institutionalizes this distinction and extensions points are the way how *public* methods are made *published* methods.

Now we are in a position to explain in more details what a plug-in is and what it contains. Each plug-in (Eclipse, 2007):

- Contributes to one or more extension points;
- Optionally declares new extension points;
- Depends on a set of other plug-ins;
- Contains Java code libraries and other files;
- May export Java-based APIs for downstream plug-ins;
- Lives in its own plug-in subdirectory;

How different plug-ins link to one another to expand the Eclipse platform, can be viewed in Figure 60. First, the "host" plug-in (Plug-in A) declares an extension point P in the Manifest file. Certainly this declaration has no interest for Java code, being used only for bookkeeping purposes by the Platform itself - and to guide developers of extending plug-ins. Therefore, it is also necessary to declare an interface (Interface I) that specifies in Java what methods the plug-in extension expects to be implemented by the extender plug-in.

The "extender" plug-in (Plug-in B), in turn, implements Interface I with its Class C – in other words, it contributes the Class C to the extension point P. During initialization time, the Platform reads all manifest files and keeps an "image" of them in a registry. This registry can then be accessed programmatically by the Plug-in A to discover which plug-ins are extending its extension points. In this example, it will discover that Plug-in B contributes Class C, will instantiate C and call its I methods.

A more detailed explanation of the Eclipse extension point mechanism will be presented in the next chapter, when I will describe how this facility has been used to keep the model plug-in and the editor plug-in synchronized.

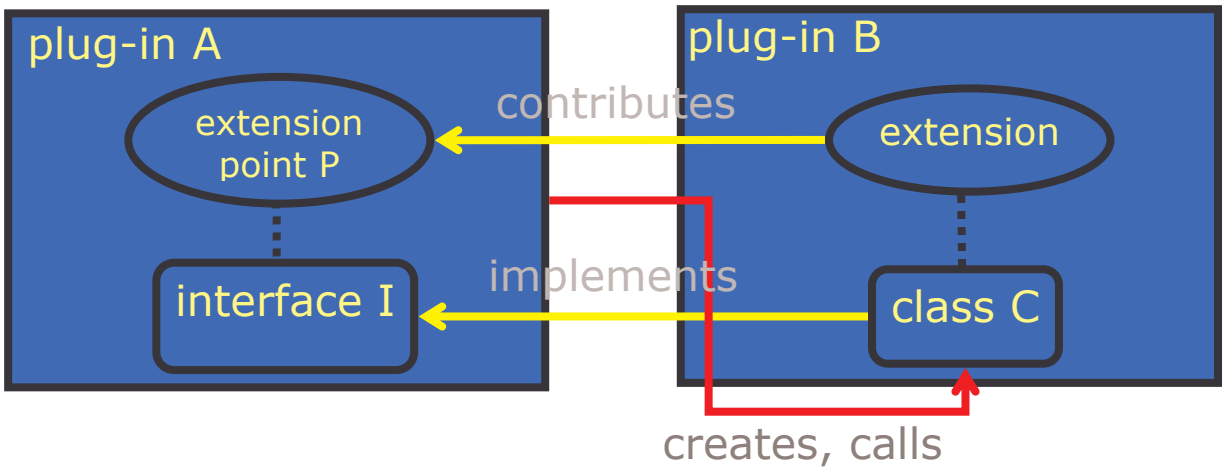


Figure 60: Extending plug-ins. (Source: Eclipse, 2007)

Most of the tasks performed by the Platform Runtime take place during initialization. On start-up, the Eclipse Platform discovers all the available plug-ins – usually located in a determined directory - and matches extensions with their corresponding extension points. As mentioned above, also at start-up, the Platform Runtime builds a global plug-in registry and caches it on disk.

In order to reduce loading time – which usually takes very long in plug-in based applications – a plug-in is only activated when its code actually needs to run (for example, when the user selects a menu item contributed by the plug-in). This strategy also affords scalability to platforms with a large base of installed plug-ins.

Resources: Provides a layer on top of the operating system resources (files and folders, including linked resource locations). Although Eclipse tools are able to work with ordinary files and folders, they rather do it through an API that offers to plug-ins high-level concepts like *resources*, *projects*, and *workspace*. A **resource** is the Eclipse representation of a file or folder. A **project** maps to a user-specified folder in the underlying file system and encompasses the resources. A project can also be applied a "semantic" tag called a "nature", becoming then associated to specific behaviors. For example, the Java nature is used to indicate that a project contains the source code for a Java program – causing all java compilation unities in the designed "source" folder to be automatically compiled, for example. A **workspace** represents the user's virtual project container.

Graphical Interface: Provides a layer on top of operating systems widgets. It is composed of three main elements: the *Standard Widget Toolkit* (MacLeod & Northover, 2001; Northover, 2001; Scarpino, 2005), the higher-level JFace toolkit (Scarpino, 2005) and the "main window", called the *Workbench***.**

All Eclipse GUI components are based on **SWT** widgets, which "translate" the host operating system's visual elements to the plug-ins. As view-related elements, SWT widgets have no knowledge of application's domain objects, so **JFace** acts as a bridge between low-level SWT widgets and domain objects, performing a typical *controller* role from the MVC pattern[2].

The **workbench** is composed mainly of *views* and *editors*, which offer a "context" where SWT elements are arranged. As an example, the **Navigator** view displays workspace's projects, folders, and files. We should note that, it shows always the Resource "model", and this contrasts with other kinds of view that display "dynamic" information: for example, the standard **Properties** view "tracks" the current selection from other views and displays the properties of the selected object in these views. Also, we can list **actions** among the components of the Eclipse Workbench. Eclipse actions are pieces of code that can be called from buttons and menu items:

> When a toolbar button or a menu item is clicked or when a defined key sequence is invoked, the `run` method of some action is invoked. Actions generally do not care about how or when they are invoked, although they often require extra context information, such as the current selection or view (Arthorne & Laffra, 2004).

That is, *actions* keep the code that responds to user initiated events. They can be declared directly in the code or contributed to the workbench declaratively in `plugin.xml`, in which case they defer the actual work to an *action delegate*.

A.3.2 Eclipse as a multi-view platform

One distinguishing feature of Eclipse is that it is by nature a "multi-view" platform, when it proposes several views for the same object. It enables to explore, for example, the object's hierarchical constitution (**Outline** view); its inner constituents (**Properties** view); if it presents any problems (**Error log** view, **Problems** view), or any other aspect we might find relevant, even if for this we may have to create our own ad hoc view. Eclipse also presents the

idea of "perspective" by grouping together a subset of editors, views and actions. That is, in Eclipse jargon, a **Perspective** is a layout containing any number of different editors, views and actions.

Eclipse ships with a number of default perspectives (Resource, Java, Debug, etc.) that can be customized, or you can create completely new perspectives. For example, the **Resource** perspective presents, among others, the Navigator view – which, as mentioned before, is used to display the workspace (i.e. the projects and their resources). The **Java** perspective, in turn, replaces the Navigator view for the **Explorer** view, which, compared to the first, stresses the java “model” underlying the compilation units (e.g. java source files are presented as trees containing classes, methods, constructors, etc). Also, the **Debug** perspective, compared to the Java perspective, decreases the size of the main source window in favor of views displaying variables, breakpoints, and other debugging information. In sum, different perspectives reflect different tasks (i.e. debugging, coding, accessing resources, etc) and mold the workbench correspondingly, making it easier to the user perform the task.

Appendix B: The Eclipse Modeling Framework

B.1 Introduction

Having different views for the same "model" or grouping these views into perspectives, as shown in Appendix A, is only one aspect of Eclipse when it comes to work with models. Eclipse is equipped with its own metamodeling framework, called Eclipse Modeling Framework (EMF), which allows us to create new metamodels, thus offering a foundation for the support of integration of applications at the data level:

> While Eclipse provides a powerful platform for integration at the UI and file level, EMF builds on this capability to enable applications to integrate at a much finer granularity than would otherwise be possible (Budinsky et al., 2003).

As explained in Section 4.7.2, EMF consists of three fundamental pieces:

- The Ecore metamodel;
- A metamodel conversion framework;
- A code generation framework.

Each of these pieces is described in the coming section.

B.2 Components of the Eclipse Modeling Framework

B.2.1 The Ecore Metamodel

Ecore is an open source implementation of the Meta Object Facility (MOF), a cornerstone of the Model Driven Approach, as seen in Chapter Four. More precisely, EMF is an implementation of the Essential MOF (or EMOF), a MOF subset that corresponds to the constructs found in object oriented programming languages (i.e. classes, operations, properties, parameters, etc.). Please refer to (Gerber & Raymond, 2003) for a more in-depth

analysis of the differences between MOF and EMF. A class diagram illustrating Ecore's elements is reproduced in Figure 61.

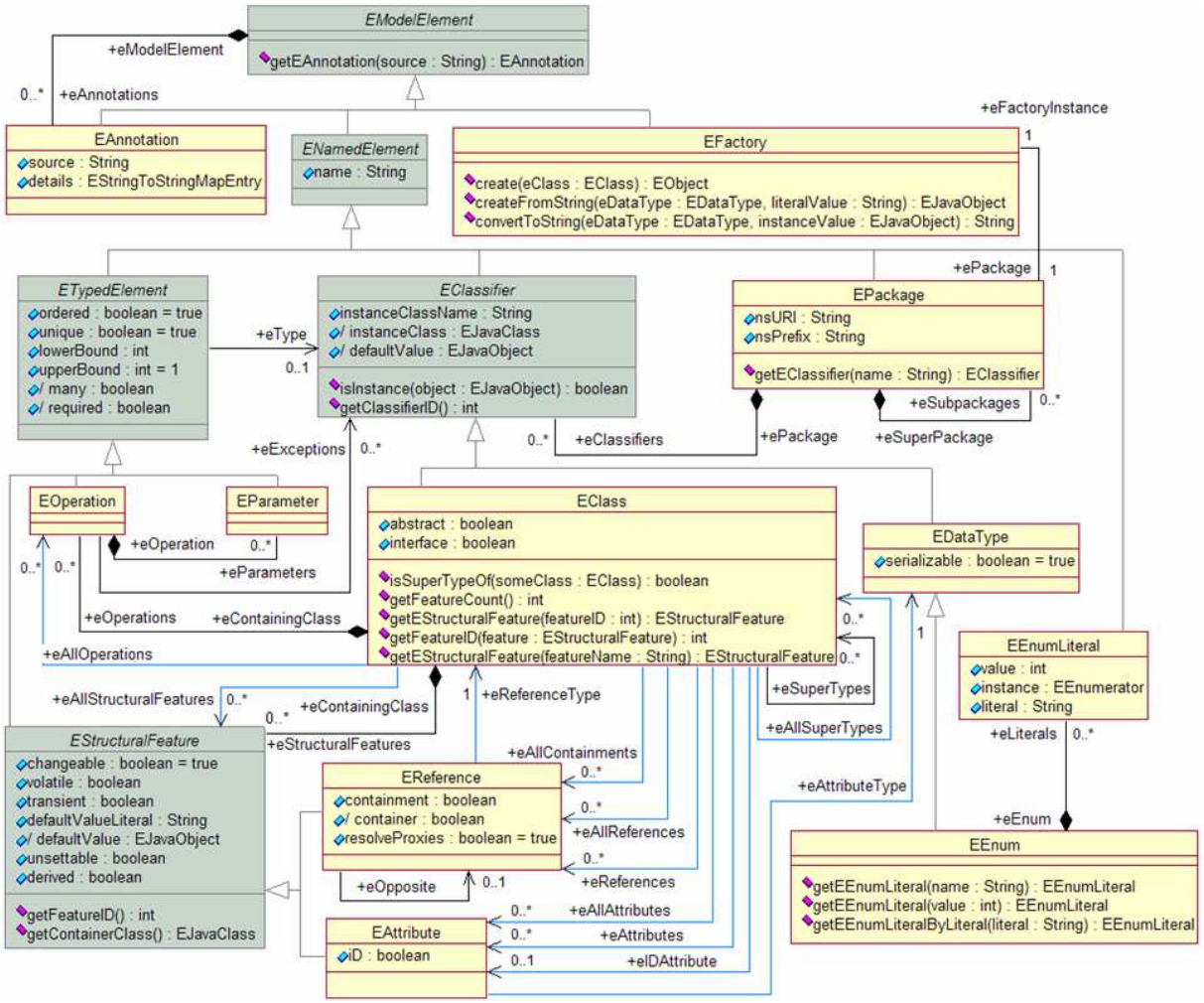


Figure 61: the Ecore metamodel

It can be noticed that the Ecore elements use about the same UML terms, to which it prepends the letter "E", and bear equivalent semantics to UML's corresponding constructs. Programmatically, each element is represented by a Java class. Take, for example, the following Ecore classes:

- **EClass** is the (meta-)class that represents classes themselves. It has a name, zero or more attributes and zero or more references. It can refer to zero or more supertypes, thus offering an inheritance mechanism.
- An **EAttribute** is used to represent a modeled attribute and has a name and a type.

- An **EReference** is used to represent one end of an association between classes. It has a name, a boolean flag to indicate if it represents containment, and a reference (target) type, which is another class. EReferences replace UML associations.
- An **EDataType** is used to represent the type of an attribute, acting as "delegates" to Java's primitive types, like `int` or `float` or well-known object types like `java.util.Date`.

We can now use instances of the classes above to describe the class structure of our application model - or **core model**, in the EMF jargon. For example, in our *learning strategy* model, we would have a **Student** class as an instance of EClass - named "Student" – and a **Name** attribute for the Student class using an EAttribute class (called "studentName"). Then we could associate both classes by adding **studentName** to the list of attributes of **Student** (`eAttributes`). The objects representing this core model can be seen in Figure 62:

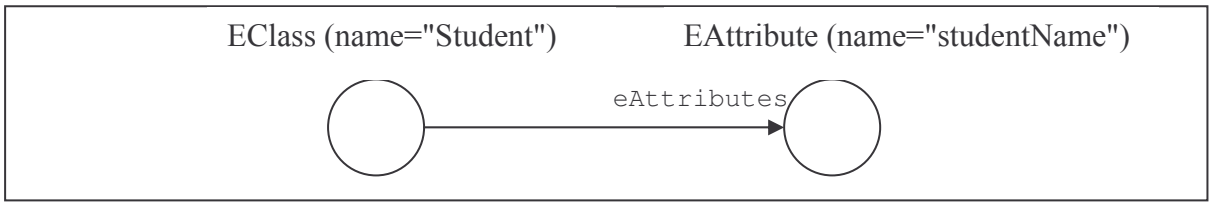


Figure 62: the *learning strategy* core model

If now we want to store the *learning strategy* model above, we have to choose a serialization method. EMF proposes the XML Metadata Interchange (XMI) as the canonical representation for Ecore models. Serialized as XMI, our model will look something like the text contained in Listing 19. There we see that it uses the elements' supertypes as references (i.e., `eClassifiers` instead of `eClasses` and `eStructuralFeatures` instead of `eAttributes`), but the structure is the same – i.e. Figure 62 and Listing 19 represent about the same conceptual model; only the references have changed.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="strategy">
  <eClassifiers xsi:type="ecore:EClass" name="Student">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="studentName"/>
  </eClassifiers>
</ecore:EPackage>
```

Listing 19: the learning strategy model

There are a few distinct forms to create a core model like the one above:

1. Directly using the EMF editors: here you can use either the unadorned tree-based sample editor provided with the EMF or a graphical editor like the Ecore diagram provided as an example by the GMF project[49].
2. Writing XMI file directly: Certainly, this error-prone method should not be considered but for very simple models.
3. Exporting from external contributions: i.e. editors that let you handle your models in any graphical notation and that, even if presenting its own native format, allow you to save in the Ecore format as well. An example of such an editor is the EclipseUML[50] from the Omondo Company, which allows you to draw a graphical UML model and save it in the Ecore format.
4. By converting from another metamodel specification – e.g. expressed in an XML Schema or any metamodel specification for which a Model Importer is available. This topic will be discussed in the next sub-section.

Figure 63 represents each of these different ways of creating a core model (or domain model):

[49] http://www.eclipse.org/gmf/

[50] http://www.omondo.com/

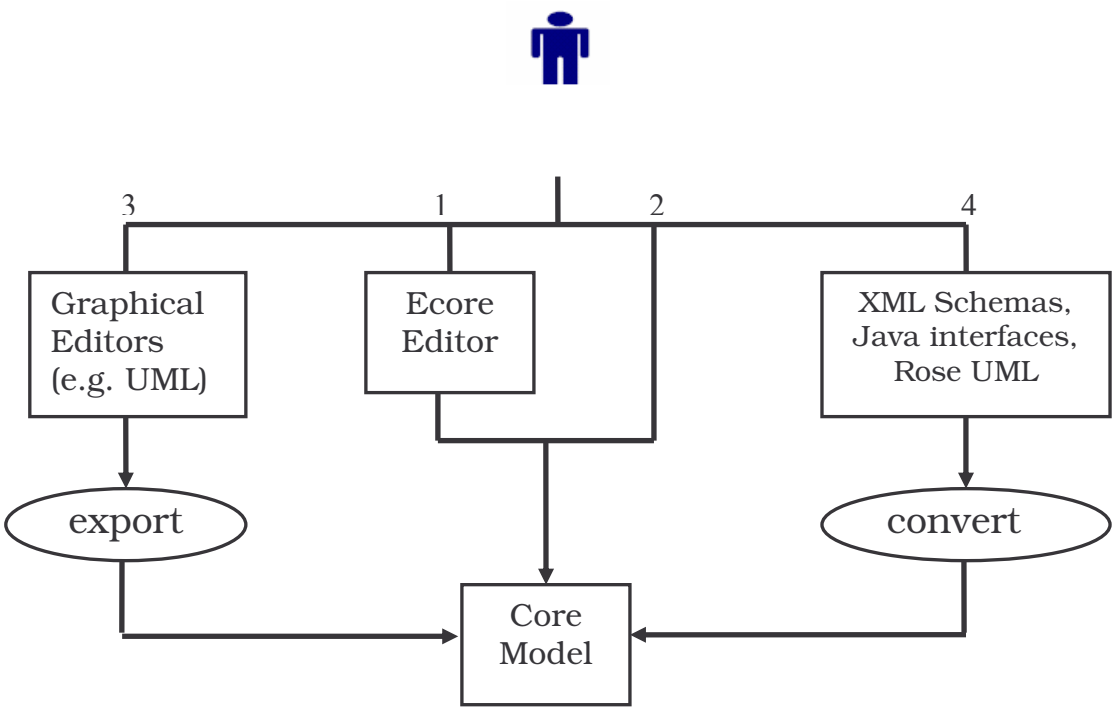


Figure 63: Creating « core » models

Once created, an Ecore model can then be used both at development time and at runtime:

> During development, the core model is the primary source of information for the EMF generator, when it produces code to be used in the application. […] At runtime, the core model is used by generic framework code to determine correct behavior for that particular model, and is likewise available to user-written code that needs to dynamically discover particulars of the model (Budinsky et al., 2003)

It is also possible to create a core model programmatically, by using a reflective API. However, a model so created resides only in memory and, consequently, will be short-lived unless we find a way of serializing it. For that, EMF provides a Resource API, which represents a persistent container for the created objects (i.e. it allows to save the objects into - and load them back from – XMI files).

B.2.2 The metamodel conversion framework

Even if a core model can be created from scratch using Ecore editors, it is common that developers already have one or more domain conceptualizations serialized in another metamodel specification, different from Ecore, what would call for reuse. If it is the case,

EMF provides tools that permit to convert from other forms of metamodel specification, what would allow developers to leverage "legacy" models. This form of creating core models corresponds to the fourth method from the previous sub-section. For this method to work it is necessary to have a Model Importer for each specification we want to convert from. Out of the box, EMF provides three Model Importers, carefully chosen so as to integrate some of today's most important technologies:

- **Java Interfaces**: EMF accepts ordinary Java interfaces incremented with Javadoc-like annotations to express model properties not captured by method declarations.
- **UML models**: (expressed in Rational Rose files): You can also import a UML model defined in Rational Rose (.mdl) files. The reason Rose has this special status is because it's the tool that has been used to 'bootstrap' the implementation of EMF itself" (Budinsky et al., 2003).However, if you prefer to use an open source free software editor, the Eclipse UML2 project[51] provides an importer for its files (.uml2 extension).
- **XML Schema**: To create an object model for manipulating an XML data structure of some type and/or already have a domain specification serialized in an XML Schema file, using the XML Schema model importer EMF is maybe the best choice. EMF can transform you Schema in a core model and subsequently into a Java API for manipulating instances of that schema. Additionally, EMF allows you to persist by default your objects as an XML instance document conforming to the Schema (although XMI is also enabled).

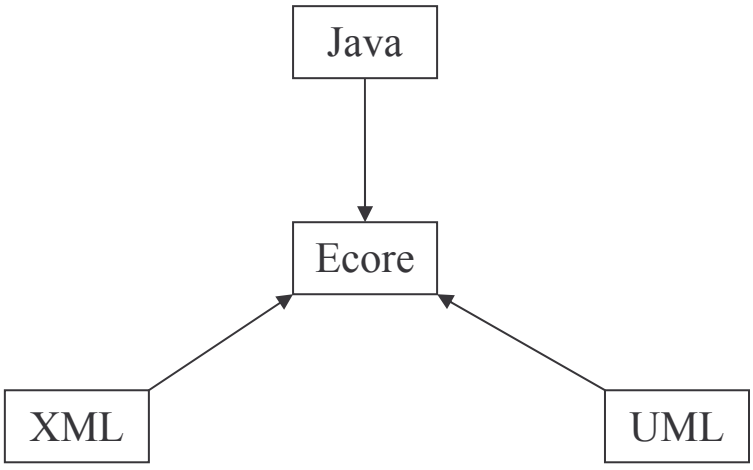


Figure 64: Converting Ecore from other metamodels

B.2.3 The code generation framework

Granted, having a core model – i.e. a domain model expressed in Ecore - is not the ultimate need of most end users. To be useful, a core model must be the origin of something users can make use of. As an implementation of the Language Oriented Programming paradigm, EMF proposes to transform these models into code that can be executed and used to address end user's needs. Better yet, this executable code should be only a few mouse clicks away from the core model. It is exactly the boost in productivity that results from a user friendly and automatic code generation what coaxes users into adopting EMF - and LOP approaches in general.

EMF uses the Java Emitter Templates (JET) technology for generating Java code of hand written quality. For a more detailed discussion of how JET works, please refer to the section 4.6.2.3. In this section I will instead focus on how JET fits within the EMF framework.

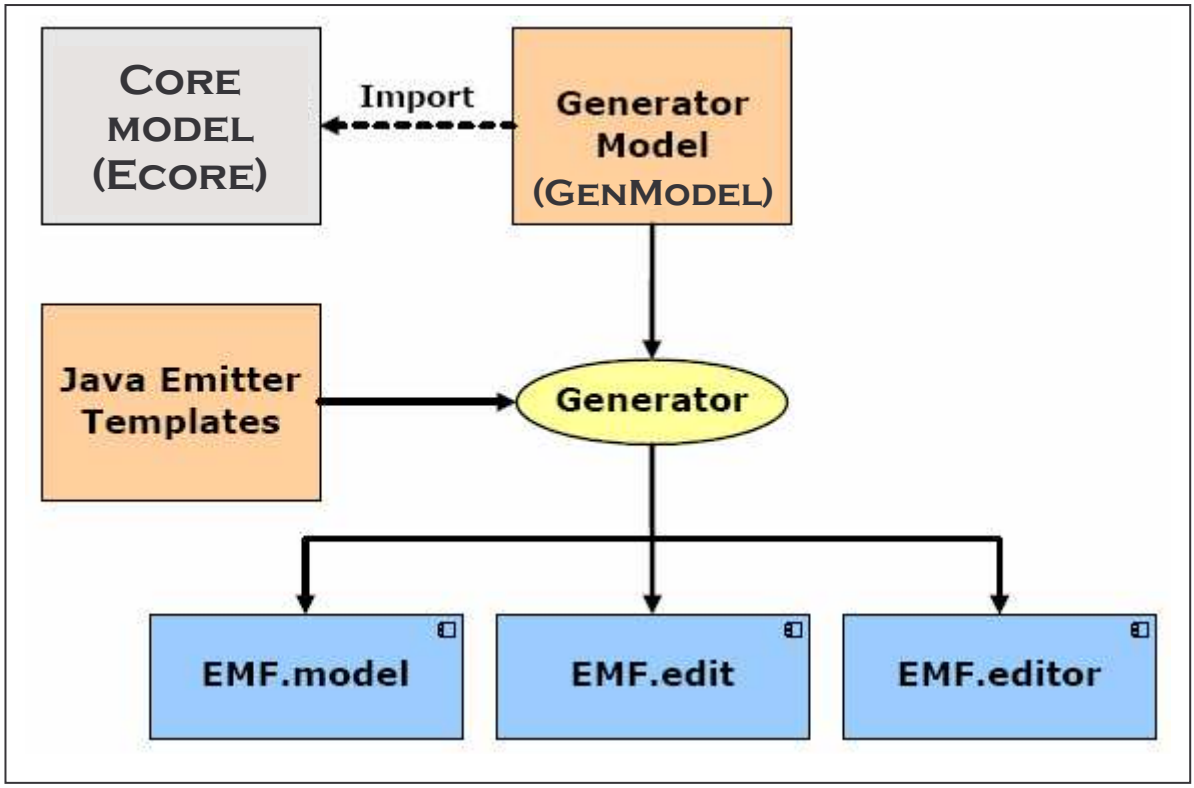


Figure 65: EMF Generator Framework

[51] http://www.eclipse.org/modeling/mdt/?project=uml2

As seen in Figure 65, the initial core model is converted into the GenModel, which contains additional information that does not constitute part of the data model itself but that is essential for code generation. Consequently, it is the generator model objects that actually drive code generation. Keeping two models was a design decision of the EMF creators:

> Separating the generator model from the core model like this has the advantage that the actual Ecore metamodel can remain pure and independent of any information that is only relevant for code generation (Budinsky et al., 2003).

Thus, each of the classes that make up the generator model decorates[52] its correspondingly named Ecore class and includes attributes for their related generator options. Genmodel conserves a similar structure as the *ecore* metamodel – only replacing the "E" prefix by "Gen". For example GenPackage objects decorate EPackage objects, GenClass decorates EClass, etc. The complete generator model as well as the relation of its elements to the corresponding *ecore* elements is shown in Figure 66:

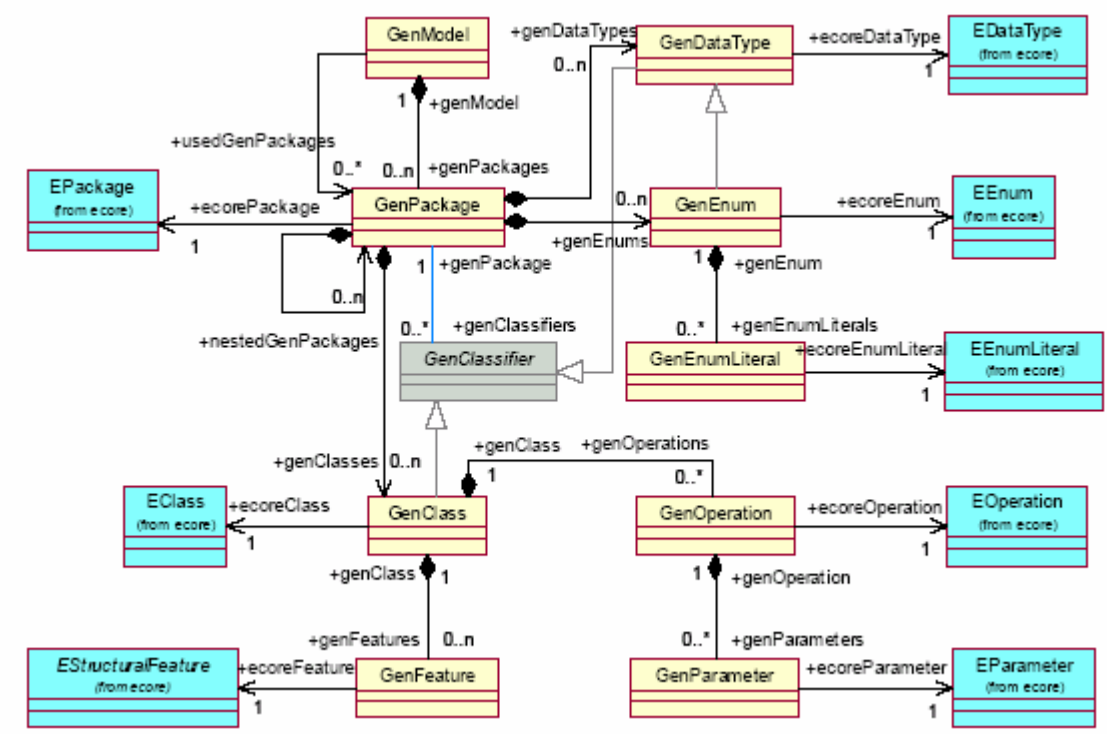


Figure 66: the Generator Model and its relation to Ecore

[52] The word *decorate* is used here in an analogy to and borrow the semantics of the decorator pattern (Erich Gamma et al., 1995).

Nevertheless, only four generator model classes create content (GenModel, GenPackage, GenClass, and GenEnum), each giving rise to a different code generation unit: model, package, class, and enum.

As explained in the last chapter, to speed up the generation process, templates can be converted into template classes (also called *static* templates) and compiled ahead of time. Template classes make call backs to GenModel objects, which return the code generation specific details either from their own attributes or from their respective core model objects.

B.3 Default Generated Code

With default JET templates, three kinds of code -- packaged by default into three different plug-ins --, are automatically generated:

1. **EMF.model** (or EMF core): in this part we have the core model expressed in Java. For example, each class from the core model gives rise to an interface and a corresponding implementation class (e.g. the core model's Student class will cause to appear a Student.java interface and its implementing class, StudentImpl.java). Thus, more than just plain bean-like objects, with their getters and setters, generated interfaces extend directly or indirectly from the base interface `EObject`, which means that generated objects present, in addition to domain based code, three main behaviors:
   - `eClass()` returns the object's metaobject (an `EClass`), which may be interesting if you want to generically access the objects instead of using the type-safe generated accessors.
   - `eContainer()` and `eResource()` return the object's containing object and resource.
   - `eGet()`, `eSet()`, `eIsSet()`, and `eUnset()` provide an API for accessing the objects reflectively.

Other important classes generated for a model are: a **factory** and a **package**. While the generated factory implements the *factory pattern* for generated objects (i.e. new objects should not be created using their constructors, but through the factory's `create` method - one for each class in the model), the generated **package** provides convenient accessors for all the Ecore metadata for the model.

2. **EMF.edit**: Given the same model definition, we can also generate the EMF. Edit, which contains *item providers* and other classes needed to edit the model. In sum, the EMF.edit is the responsible for connecting the Eclipse UI and the EMF core frameworks.

   The sample application that illustrates the use of the generated model code, described above, is an editor for the model. Certainly, as an editor, it uses Eclipse UI facilities, particularly, JFace and SWT widgets. However, when developing a functional editor, a large portion of its code is not directly concerned with the UI; so, the EMF team decided to put this part in a specific plug-in, thus allowing its reuse outside of the Eclipse UI framework. It contains facilities like a **command framework** (allows, for example, to "undo" or "redo" editions), **editing domain** support (acts as a factory for creating commands) and **item providers** (mechanism used to adapt model objects so they can provide all of the interfaces that they need to be viewed or edited – I will show an example of it in the next section).

3. **EMF.editor**: This part consists in the UI-dependent code of the sample editor. Wrapped in a plug-in of its own, it comprises, in addition to the **editor** class itself, an **action bar contributor** –where Eclipse *actions* can be added – and a **wizard** - interactive UI element that lead a user through the creation of a new model instance, using step-by-step dialogs.

Meant to be simple, once you have created a core model (like the guileless *learning strategy* model, sketched in the section 5.4.2.1 above) the process to generate a new editor resumes to selecting an item from the context menu shown below:

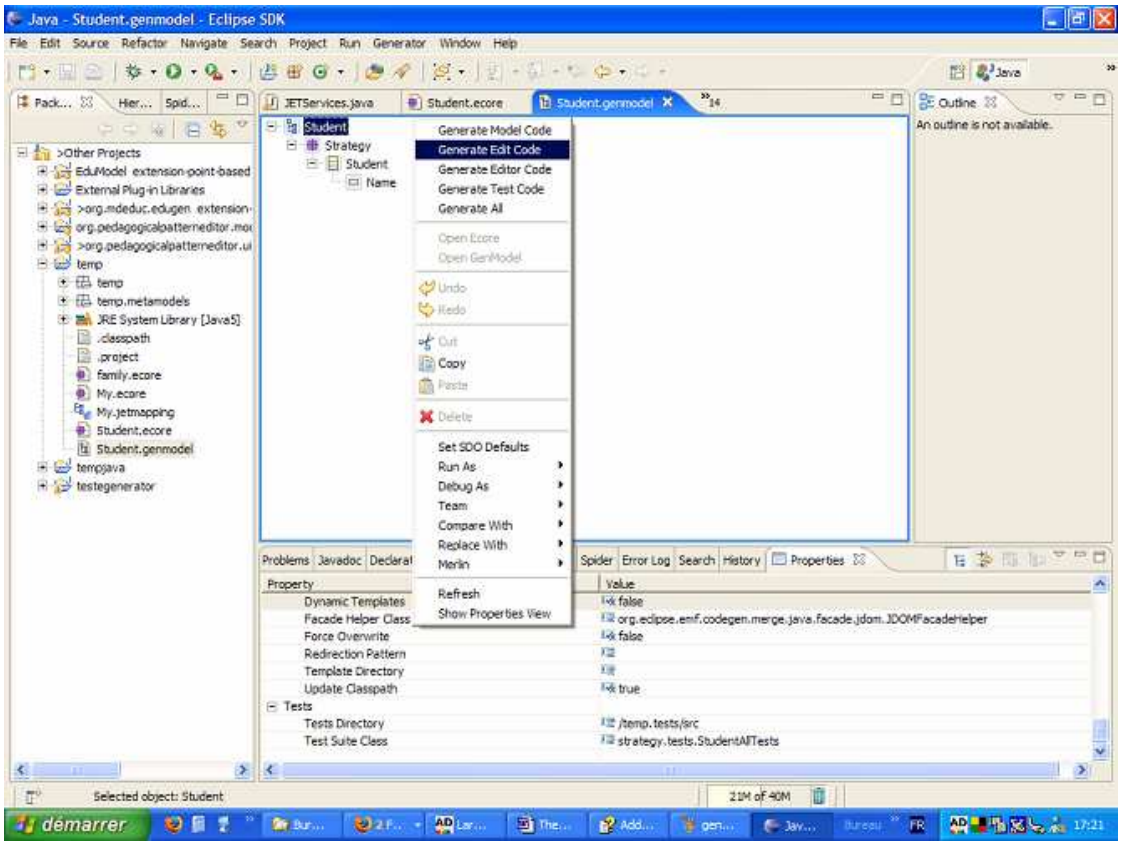


Figure 67: Generating code with EMF

As we can see in the figure, users can select to generate each of the three plug-ins separately (i.e. one for the model, one for the UI-independent part and another for the UI-dependent part of the editor) or to generate all at once (choosing the "Generate All" menu option).

Additionally, the user may wish to generate code that helps to test the generated application (selecting the "Generate Test Code" option). It will generate test cases, a test suite and a stand-alone example that uses the test classes.

We can also see in Figure 67 the Eclipse's *properties view*, from which we can have access to dozens of items permitting to personalize the generation step, what makes of this process a highly configurable one.

It is also important to no note that, even if the generated editor is fully functional, users may want to customize it for their particular needs. To do it, they will have to change the generated code. So, suppose that you spent a few hours adapting the generated code and that a few days later you decided to change the core model from which the code originated. Certainly, you

will not want to see the newly automatically generated code to overwrite the modifications you did in the old code. The generator framework offers a solution for this problem. In that case, you will have to "flag" the parts of code that have been manually edited and the generator framework will leave them untouched. That is, it will perform a merging between the newly generated editor and your customized code, thus preserving your hand written changes while changing the final application to conform to the upgraded model.

B.4 Programming with EMF

After having shown some of the features offered by the EMF framework, it is important to know how they can be used in an actual program. Thus I advance in this section some program excerpts that make use of the features previously introduced and that will give an idea of how the MDEduc prototype, described in the next chapter, has been programmed.

More than just generator tool, EMF is also a *runtime framework*, which offers programmers facilities for creating objects (Factories), accessing their metadata, serializing objects, managing containers of objects, in addition to a reflective API allowing "to work with any EMF objects, regardless of the model that defines them" and a dynamic API, which "allows you to create data without generating any code at all" (Budinsky et al., 2003). These runtime features are described in the next sections.

B.4.1 Packages and Factories

EMF generates package and factory classes that allow you to easily access a model's metadata and create instances of the modeled classes. These generated classes are subclasses of the generic Ecore classes, EPackage and EFactory.

You use factories to create instances of model objects:

Listing 20: Using the generated factory

```
StrategyFactory factory = StrategyFactory.eINSTANCE;
Student student = factory.createStudent();
```

You use Package to access metadata:

Listing 21: Using the generated package

```
StrategyPackage strategyPackage = StrategyPackage.eINSTANCE;
EClass studentClass = strategyPackage.getStudent();
EAttribute nameAttr = strategyPackage.getStudent_Name();
```

Suppose now that the StrategyPackage contains other objects in addition to Student (e.g. Instructor, Book, Objective, etc.) and you want to iterate over all of them to display their attributes (name, etc.) in an editor (so that attributes can be modified, for example). In that case, it would be interesting to have a way to access the StrategyPackage metadata generically, so you don't have to explicitly decline the name of each class. In this case, EMF provides a handful reflective API that addresses this problem conveniently. So, the Student class could be obtained like this:

Listing 22: Accessing package metadata generically

```
EClass studentClass =
  (EClass)strategyPackage.getEClassifier("Student");
```

Note that, instead of being hard-coded into the accessor method, like in Listing 21, the name of the name is passed as a parameter, which facilitates its use in a loop, for example. Note also that the EClass' superclass (i.e., EClassifier) has been used, which means that we could iterate also over other types of classifiers, like data types (EDataType) and enumerations (EEnum) (please refer to Figure 61 to see the layout of the Ecore metamodel).

Similarly, we can also create an object generically, like in the snippet below:

Listing 23: creating an object generically

```
EPackage strategyPackage =
    EPackage.Registry.INSTANCE.getEPackage(strategyPackageURI);
EClass studentClass =
(EClass)strategyPackage.getEClassifier("Student");

EFactory studentFactory = strategyPackage.getEFactoryInstance();
EObject student = studentFactory.create(studentClass);
```

Here I retrieved the package from the EMF's Package Registry, which provides a mapping from namespace URIs to EPackages. This registry is populated by plug-ins declaring the `generated_package` extension point. Optionally, a package can also be retrieved using the `Resource` API. As we saw in Listing 19, an `EPackage` is the root element in a serialized core model, which means that it is the *first* element (i.e., position zero) in a `Resource`:

Listing 24: Retrieving a package from a resource

```
ResourceSet resourceSet = new ResourceSetImpl();
URI fileURI = URI.createURI("../strategy.ecore");
Resource resource = resourceSet.getResource(fileURI, true);
EPackage strategyPackage =
(EPackage)resource.getContents().get(0);
```

B.4.2 Notifiers and Adapters

EMF generated classes, by subclassing EObject, also inherit the `Notifier` interface, which add two important behaviors:

1. It keeps a list of "observers", i.e. classes that will be notified if the object's state changes.
2. It specifies the `eNotify(Notification)` method, which will notify the observers when the change occurs.

In fact these observers are called *adapters*, because they are used to adapt, i.e. to augment the original behavior of a model object:

> As behavioral extensions, they can adapt the model objects to implement whatever interfaces the editors and views need, and at the same time, as observers, they will be notified of state changes which they can then pass on to listening views (Budinsky et al., 2003).

Thus, notifiers and adapters are fundamental mechanisms in the EMF framework since they offer a way to provide all of the interfaces that objects need to be viewed or edited. For example, if we wanted our model objects to be visible in the Properties view, we would have to add the "Properties Adapter" (actually called `IPropertySourceProvider`) to the objects' `eAdapters` list :

Listing 25: adding an adapter

```
purchaseOrder.eAdapters().add(myIPropertySourceProvider);
```

When now any change is made to the model object it will alert the Properties View, which will respond accordingly (for example, by refreshing its properties' table, causing the changed property to be updated). In fact, adapters expand the behavior of an object without having to resort to inheritance mechanisms (what would signify to "pollute" the API). This is a direct application of a design pattern called Extension Object, also known as Extension Interface (Martin, Riehle, & Buschmann, 1998) – and *not* of the Adapter Pattern, as in (Erich Gamma et al., 1995).

In reality the above explanation is an oversimplification of the actual mechanism used in EMF, which uses factories to adapt observers to model objects (justified by the fact that there can be many different adapters for the same model object). But it is enough to illustrate a mechanism that will be extensively used in the MDEduc prototype, described in the next chapter.

B.4.3 Dynamic EMF

As we saw in Section 5.5, the MDEduc prototype offers a text editor adapted to the syntax of pedagogical patterns that permits learning designers to select and capture terms from it and add it dynamically to a model – also created on the fly. This is a typical scenario that calls for a way of creating and sharing a model without having to generate a type-safe API.

EMF provides a way of creating core models programmatically. Furthermore, once created, dynamic models are treated exactly like any other core model developed in any of the methods listed in the item 5.4.2.1 above. That is, they can be persisted in resources, they can give rise to generated code, etc.

To illustrate the above explanation with an example, we could obtain, for our trivial Learning Strategy model, the same result as in Listing 19, using the following code from the dynamic EMF API:

Listing 26: learning strategy model dynamically created

```
EcoreFactory ecoreFactory = EcoreFactory.eINSTANCE;
EcorePackage ecorePackage = EcorePackage.eINSTANCE;

EClass studentClass = ecoreFactory.createEClass();
studentClass.setName("Student");

EAttribute studentName = ecoreFactory.createEAttribute();
studentName.setName("name");
studentName.setEType(ecorePackage.getEString());
studentClass.getEAttributes().add(studentName);

EPackage strategyPackage = ecoreFactory.createEPackage();
strategyPackage.setName("strategy");
strategyPackage.setNsPrefix("strategy");
strategyPackage.setNsURI("http:///edu.example.strategy.ecore");
strategyPackage.getEClassifiers().add(studentClass);
```

Appendix C: Other Language Oriented Programming implementations

C.1 Literate programming

One of the precursors of the language oriented paradigm, Literate Programming was created by Donald E. Knuth, who imagined programs as pieces of literature, conceived primarily for human consumption:

> I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: "Literate Programming". Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do (Knuth, 1992).

The idea is for programmers to write files containing the program description - in a rather documentational style – with pieces of code injected into it and then have tools reconstitute both documentation and code – in any language. As (Williams, 2000) explains:

> The "program" then becomes primarily a document directed at humans, with the code being herded between "code delimiters" from where it can be extracted and shuffled out sideways to the language system by literate programming tools.

A diagrammatic view of the above explanations is found in Figure 68.

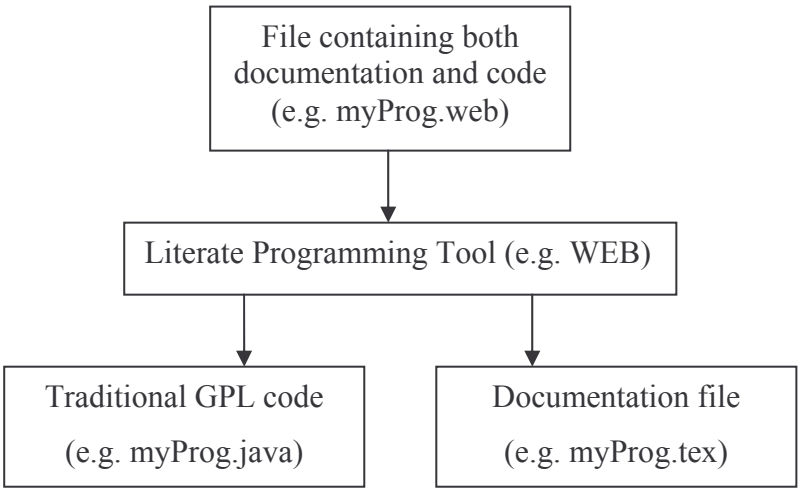


Figure 68: Literate Programming schema

Knuth also created the first environment for Literate Programming, called WEB. It consisted of two processors, called WEAVE and TANGLE, each one acting on a WEB document in a different way. WEAVE "weaves" the document for a human reader, producing an output file with a .tex extension. This file can then be fed to the T_EX typographic tool, yielding a “pretty printed” version of the program’s documentation. TANGLE in turn, produces a plain programming language file to be compiled, linked and executed (Knuth & Levy, 2002).

Present Status

Literate programming is supported by many tools, all of which provide some way for authors to interleave program source code with well typeset documentation. A search in the SourceForge software repository yields eighteen projects ranging from scripts to full fledged editors allowing you to write a document describing your software and extract the code for your software from that document. Some of these projects have tens of thousands and even hundreds of thousands of downloads, indicating that there are active communities behind them.

No evidenced of Literate Programming based applications for the domain of learning design – and not so for the ampler e-learning domain – has been found.

C.2 Intentional Programming

If the importance of abstraction in the reading and writing of programs is something taken for granted in present days, Intentional Programming will enforce this principle, by enabling programmers to express the *intention* they have in mind without having to specify the details of how the intention will be executed.

(Simonyi, 1996) chose the *intention* metaphor to reflect the fact that programmers should stay focused on their original intention, while avoiding being distracted by complex general purpose language syntax details:

> From the programmer's point of view, intentions are what would remain of a program once the accidental details, as well as the notational encoding (that is the syntax) had been factored out. Intentions express the programmer's contribution and nothing more. The name "intention" suggests that they express directly the programmer's computational intent.

Similarly to all other LOP approaches, Intentional Programming intends to isolate human programmers from the burdensome task of writing pieces of software – a task which would then be left to code generators. However, telling the generator what program to write would be accomplished through an interface that would accept, instead of general purpose languages, models and DSLs to describe the program. Figure 69 illustrates this process.

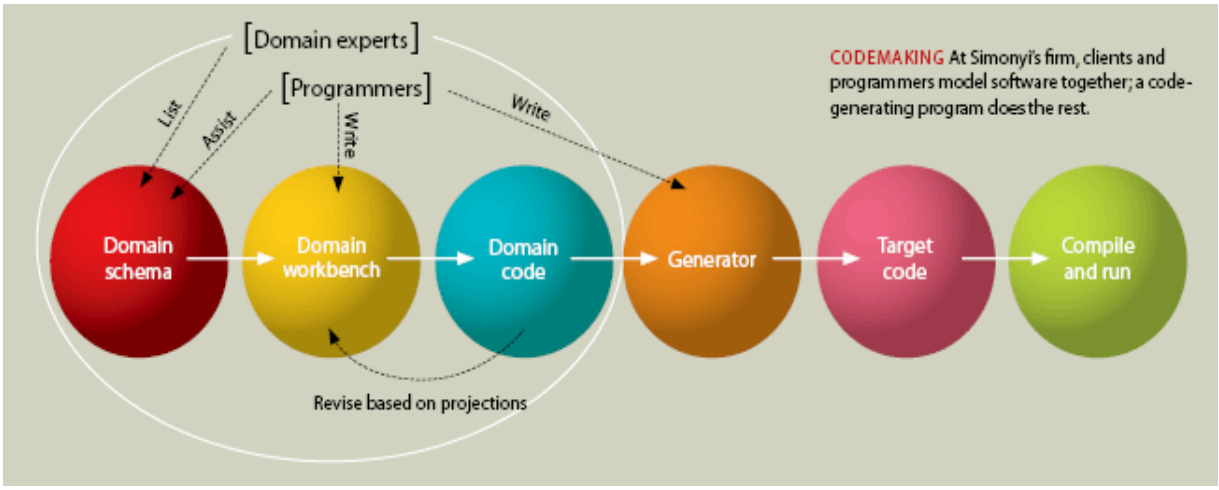


Figure 69: Intentional Programming (source: Technology Review online magazine)

The process starts with a **domain schema** – where the different concepts of the domain are defined –created by domain experts. The domain schema is then fed into the **domain**

workbench, which includes an editor permitting to change it and then generate the **domain code**, in a uniform format called an intentional tree.

Another central piece in this architecture is the code **generator,** which isolates the concerns of the domain experts from that of programmers. It takes in a domain code and generates the **target code** (i.e. the target code is a mere function of the domain code). Generators are intended to be "coded", that is, programmers still have to do programming, using traditional software engineering techniques (agile methods, design patterns, etc.); in this case, "[t]he savings are achieved when the domain code is maintained and the generator is re-run" (Simonyi, Christerson, & Clifford, 2006).

In Simonyi's own words, Intentional Programming will be to software creators what the WYSIWYG technique[53] is to word processors' users.

> Wysiwyg editors simplified document creation by separating the document contents from the looks and by automating the reapplication of the looks to changing contents. In the same way Intentional Software simplifies software creation by separating the software contents in terms of their various domains from the implementation of the software and by enabling automatic regeneration of the software as the contents change (Simonyi et al., 2006).

Present Status

Contrary to the other LOP implementations, Intentional Programming still have not released any product that puts its principles into practice. For the time being, all there is to see about this approach are a few papers describing its principles.

C.3 Generative programming

First proposed in the PhD thesis from (Krzysztof Czarnecki & Eisenecker, 2000), generative programming is a style of computer programming that uses automated source code creation to improve programmer productivity. It uses techniques as distinct as generic classes, templates, aspects, and code generators. Quoting its own author, generative programming is:

> A software development paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge (Krzysztof Czarnecki & Eisenecker, 2000).

It advocates, then, an industrial manufacturing culture for software production, as opposed to the one-of-a kind approach proposed by traditional software engineering methods. According to the generative programming philosophy, the transition to the automated manufacturing in software requires two steps (Czarnecki & Eisenecker, ibidem):

- Moving the focus of programming from the engineering of single systems to the engineering of **families** of systems: this means that software components should be designed to meet a common product line architecture, enabling its reuse among products of the same family.
- Automating the assembly of the implementation components using software **generators**: that is, the direct producer of a program is not a programmer, but another software program.

In the generative programming conception, software should be generated based on a common generative domain model, composed of three parts: the problem space, the solution space and the configuration knowledge, which specifies how to translate abstract requirements into specific constellations of components. Furthermore, the configuration knowledge should be kept in a programmatic form - and not in the programmers' brains. Figure 70 illustrates the generative domain model:

---

[53] The WYSIWYG technique is also credited to Simonyi, as he was the researcher responsible for Bravo, the first word-processor to show on-screen exactly how a document would look in print.

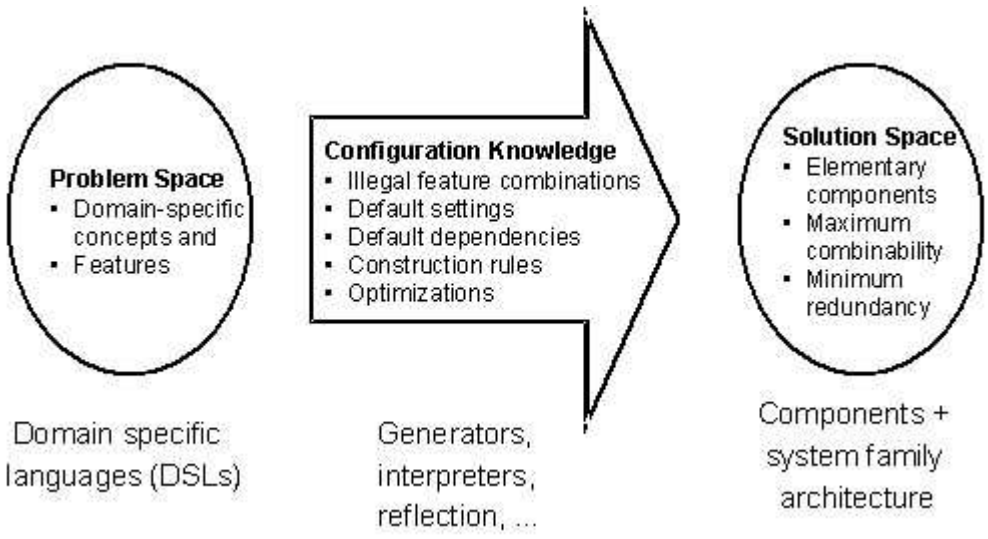


Figure 70: Elements of a generative domain model (source: Czarnecki & Eisenecker, 2000)

To meet the Generative Programming standard, an application should observe a few principles (K. Czarnecki, Eisenecker, Gluck, Vandervoorde, & Veldhuizen, 2000):

Separation of concerns: this term refers to the importance of dealing with one important issue at a time. To avoid program code which deals with many issues simultaneously, generative programming aims to separate each issue into a distinct set of code. These pieces of code are combined to generate a needed component.

Parameterization of differences: allows us to compactly represent families of components (i.e. components with many commonalities).

Analysis and modeling of dependencies and interactions: "Not all parameter value combinations are usually valid, and the values of some parameters may imply the values of some other parameters". These dependencies are referred to as *horizontal configuration knowledge*, since they occur between parameters at the same level of abstraction.

Separating problem space from solution space: The problem space (i.e. the domain-specific abstractions) and the solution space (i.e. the implementation components) have different structures and are mapped to each other with *vertical configuration knowledge*. The term *vertical* refers to interaction between parameters at different abstraction levels. Both horizontal and vertical configuration knowledge are used for automatic configuration.

Eliminating overhead and performing domain-specific optimizations: As we have seen in the example of the generator iterative approach, by generating components statically (i.e., at generation time), much of the overhead due to unused code, run-time checks, and unnecessary levels of indirection may be eliminated.

Present Status

The fact that Generative Programming is a common expression, which lends itself to many different purposes, does not help us trace which of the existing products are implementations of Czarnecki's ideas and which are not. For example, a product offering, exclusively, the *template* and *code generator* tandem should not qualify for displaying the Generative Programming label (and yet some do it, e.g. Jostraca[54]). In any case, there is always the original AmiEddi program, made available at the Generative Programming official site[55], to illustrate the principles of this approach. Unfortunately it apparently has no new version released since 2001. Also, no application developed with the Generative Programming technique and specific for education domain has been found.

[54] http://www.jostraca.org/

[55] http://www.generative-programming.org/

Appendix D: the complete specification of the “What is Greatness” learning scenario (XMI file)

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="whatisgreatness"
   nsURI="" nsPrefix="">
 <eSubpackages name="entity" nsURI="" nsPrefix="">
   <eSubpackages name="users" nsPrefix="">
    <eClassifiers xsi:type="ecore:EClass" name="Student">
     <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
      <details key="stereotype" value="Entity"/>
     </eAnnotations>
     <eOperations name="findByLoginAndPassword" eType="#//entity/users/Student">
      <eParameters name="login" ordered="false" eType="ecore:EDataType
       http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eParameters name="password" ordered="false" eType="ecore:EDataType
       http://www.eclipse.org/emf/2002/Ecore#//EString"/>
     </eOperations>
     <eStructuralFeatures xsi:type="ecore:EAttribute" name="email"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
     <eStructuralFeatures xsi:type="ecore:EAttribute" name="firstName"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
     <eStructuralFeatures xsi:type="ecore:EAttribute" name="lastName"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
     <eStructuralFeatures xsi:type="ecore:EAttribute" name="login"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
     <eStructuralFeatures xsi:type="ecore:EAttribute" name="password"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Teacher">
     <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
      <details key="stereotype" value="Entity"/>
     </eAnnotations>
     <eStructuralFeatures xsi:type="ecore:EAttribute" name="login"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
     <eStructuralFeatures xsi:type="ecore:EAttribute" name="password"
       eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
   </eSubpackages>
 </eSubpackages>
```

```
<eSubpackages name="session" nsURI="" nsPrefix="">
  <eSubpackages name="students" nsURI="">
    <eClassifiers xsi:type="ecore:EClass" name="EnterInitialThoughts">
      <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
        <details key="stereotype" value="Screen"/>
      </eAnnotations>
      <eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
        <details key="documentation" value="initial thoughts"/>
      </eAnnotations>
      <eOperations name="enter">
        <eParameters name="thoughts" eType="ecore:EDataType
          http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      </eOperations>
      <eStructuralFeatures xsi:type="ecore:EReference" name="readContents"
          eType="#//session/students/ReadContent"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="searchInternet"
          eType="#//session/students/SearchInternet"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="discussForum"
          eType="#//session/students/DiscussForum"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="reportSubmission"
          eType="#//session/students/ReportSubmission"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="DiscussForum">
      <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
        <details key="stereotype" value="Screen"/>
      </eAnnotations>
      <eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
        <details key="documentation" value="http://greatness.forumactif.com"/>
      </eAnnotations>
      <eOperations name="login"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="lastName"
          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="firstName"
          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="email"
          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="login"
          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="readContents"
          eType="#//session/students/ReadContent"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="enterInitialThoughts"
          eType="#//session/students/EnterInitialThoughts"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="searchInternet"
          eType="#//session/students/SearchInternet"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="ReadContent">
      <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
        <details key="stereotype" value="Screen"/>
      </eAnnotations>
```

```
      <eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
        <details key="documentation" value="greateness.doc"/>
      </eAnnotations>
      <eOperations name="access"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="discussForum"
          eType="#//session/students/DiscussForum"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="SearchInternet">
      <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
      <eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
        <details key="documentation" value="http://google.com"/>
      </eAnnotations>
      <eOperations name="search"/>
      <eStructuralFeatures xsi:type="ecore:EReference" name="enterInitialThoughts"
          eType="#//session/students/EnterInitialThoughts"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="ReportSubmission">
      <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
        <details key="stereotype" value="Screen"/>
      </eAnnotations>
      <eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
        <details key="documentation" value="report"/>
      </eAnnotations>
      <eOperations name="report"/>
    </eClassifiers>
  </eSubpackages>
  <eSubpackages name="teachers" nsURI="">
    <eClassifiers xsi:type="ecore:EClass" name="CreateForum">
      <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
        <details key="stereotype" value="Screen"/>
      </eAnnotations>
      <eOperations name="create"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="ParticipateForum">
      <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
        <details key="stereotype" value="Screen"/>
      </eAnnotations>
      <eOperations name="login"/>
      <eOperations name="logout"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="submitContent">
      <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
        <details key="stereotype" value="Screen"/>
      </eAnnotations>
      <eOperations name="submit"/>
    </eClassifiers>
  </eSubpackages>
 </eSubpackages>
</ecore:EPackage>
```