1. Introduction

This first chapter of the dissertation intends to introduce DSLs. First, DSLs are generally contextualized within the process of software development. After that, the wide-ranging advantages of working with DSLs within a metamodelling approach are considered. At last, the goals of the research presented in this dissertation are stated , as well as the demonstration case and, finally, the roadmap of the document.

1.1. Software Development with DSLs

Raising the level of abstraction for software engineers to write applications is still an undergoing issue. So, models are the basis of today's Software Engineering scenario. A model is a view of a system in a multiview perspective [Zhang, *et al.*, 2005]. Using UML (Unified Modelling Language) [OMG, 2007b] to describe systems in a multiview perspective is ideal when it comes to complex systems.

Models, modelling and model transformation are the basis of MDD (Model-Driven Development) [Atkinson and Kühne, 2003; Balmelli, *et al.*, 2006]. Sendall and Kozaczynski [2003] referred to model transformation as being the process of transforming one or more source models into one or more target models following a set of transformation rules. Activities like *reverse engineering*, *application of patterns* or *refactoring* use model transformations.

MDD defines application implementations using models instead of programming languages exclusively [Atkinson and Kühne, 2003]. According to Demir [2006] and also to Atkinson and Kühne [2003], MDD makes sense when talking about models and code

generation from models. MDD was conceived for teams with more than twenty people, therefore, MDD is adequate to large-scale software industries. Its goals are to automate repetitive tasks and avoid architectural degradation, as Bettin [2004] called it. MDA (Model-Driven Architecture) [Miller and Mukerji, 2003] and SF (Software Factories) [Greenfield and Short, 2003] are two approaches to MDD. MDA is a standard of OMG (Object Management Group) [OMG, 2007a] and has UML as a major part of the approach. SF is an approach from Microsoft. It integrates domain-specific modelling languages, a kind of DSL (Domain-Specific Language) [Mernik, *et al.*, 2005], with the SPLs (Software Product Lines) [Greenfield and Short, 2003] concept born within the Software Engineering Institute of Carnegie Mellon University [Frankel, 2005].

DSLs exist since the fifties of last century [Mernik, *et al.*, 2005]. Sprinkle and Karsai [2004] argued that DSLs may have been in the base of programming languages proliferation. They are languages that use domain concepts and can be compiled in order to give birth to source code [Batory, *et al.*, 2002]. Mernik, *et al.* [2005] defined DSLs as artefacts to be used to work on a problem situated in specific domain. Sprinkle, *et al.* [2001] mentioned the need to have a language to describe specific problems. Even use case diagrams can be classified orthogonally into functionality use case diagrams (*send alert* or *receive external information*) and domain use case diagrams (*healthcare* or *education*) [Machado, *et al.*, 2005a]. In order to make general-purpose modelling languages usable in various application domains, domain-specific model engineering introduces DSLs as modelling languages [Schleicher and Westfechtel, 2001]. SDL (Specification and Description Language) is a domain-specific visual modelling language for specifying telecommunications protocols using telecommunications-related concepts [Cook, 2004]. SQL (Structured Query Language) is an example of a DSL for querying relational databases and HTML (HyperText Markup Language) an example of a DSL for writing hypertext [Bragança, 2003]. Shortly, DSLs are languages specifically targeting a particular domain, like finances, telecommunications and others. They can be either model-based or code-based. On the opposite side of DSLs are GPLs (General Programming Languages) [Batory, *et al.*, 2002; Bragança, 2003; Mernik, *et al.*, 2005; Sendall and Kozaczynski, 2003], languages which can be applied to different domains. Java is a language used to implement applications in different domains, like those mentioned earlier (Java is a GPL).

It is essential to consider software development methodologies based on models in the current Software Engineering scenario. The reason for this is that models *per se* are useless; they must be contextualized within a methodology. If organizations describe their computer-based systems using UML, then they will be aware that UML is only about notation and that

methodologies to develop models are one step ahead. A quick example of such a methodology is the RUP (Rational Unified Process), from IBM, that describes a set of best practices for MDD and goes beyond notation [Brown, *et al.*, 2005b]. RUP is a procedural guideline of software development which adopts models.

When handling DSLs, it is important to keep in mind that software evolves over time and that software evolution takes place when the DSL changes. For instance, if new requirements demand for the introduction of new domain concepts into the DSL, the DSL is going to suffer a change and, thus, software will evolve from its previous state into a new one which will consider the new domain concepts ultimately in the implementation of the software solution.

Before a methodology can be applied, modelling languages can be defined based on UML, including DSLs as DSVLs[1] (Domain-Specific Visual Languages) [Sprinkle and Karsai, 2004]. UML is still one of the widely adopted standard languages for modelling in general and, hence, it can be considered when metamodelling, or defining, the DSL.

1.2. The Advantages of Using DSLs Embedded in a Metamodelling Approach

The important thing is to solve complex problems in easier ways. Models are a strategy to accomplish this. Coordination of different teams working on different components of a wider system is a difficult task. Dealing with a change in the development of a software solution is hard. Software development methodologies based on models help to solve these issues. In fact, development based on models is a possibility to increase productivity in organizations and, consequently, reduce time-to-market by decreasing development time. Using models is also synonym of greater involvement of the customer in the analysis phase of software systems development [Machado, *et al.*, 2005a]. A use case diagram is an ideal artefact to capture the requirements of a system not only by the software engineer but also by the customer. Brown, *et al.* [2005a] stated that models are an instrument for engineers to reason about the system without delving into technological details and that they are also powerful when it concerns communicating between the various stakeholders. Models and their visual notations are more comfortably assimilated than pure code [Ardis, *et al.*, 2000; Atkinson and Kühne, 2003]. Consider refactoring as the activity that allows maintaining software applications changing its internal structure and behaviour without changing its external behaviour. Refactoring hot spots, as well as refactoring impacts, are better determined based

[1] Nowadays the term Domain-Specific Modelling Language is replacing the term DSVL.

on a graphical notation [Zhang, *et al.*, 2005].

Development productivity and quality are enhanced with modelling tools capable of automatically transform models by means of a model transformation language, both to perform default transformations but also to define new customized ones. If transformation rules are written in a GPL, developers are not expected an additional expertise to write transformation rules. On the other hand, GPL lack higher abstractions to write those transformation rules.

The use of SF allows the development of complex software of high quality, on time, on budget and with customer satisfaction. Encapsulation allows raising the level of abstraction and reduces complexity-related issues. SF aim at reaching a wider range of customer's needs with a wider software scope. A product family realizes an economy of scope, which means that multiple related designs are produced, and is targeted at custom markets where each product is unique [Greenfield and Short, 2003]. Models can be reused across a number of applications in a product family. This is a strategy of long-term investment based on improving the quality of software design [Bettin, 2004].

Targeting a specific application domain, the use of specialized constructs by DSLs has considerable advantages [Mernik, *et al.*, 2005]: (1) Greater power of expressiveness and easiness of use than GPLs'; (2) Increased productivity and reduced maintenance costs (for instance, through the specification of product line members' common architecture); (3) Reduced extent of expertise needed to manipulate the language (both domain and programming expertise); (4) GPL programming task automation (code can be generated model designed with DSLs).

The gap between the problem domain and the solution domain, in terms of abstraction, is reduced when using DSLs and domain concepts [Demir, 2006]. Above all, DSLs allow the quick creation and maintenance of a complex system [Sprinkle and Karsai, 2004].

Metamodelling is an approach to model complex systems (complex systems are characterized by various user profiles, various functionalities, various data structures) by using abstraction as a means that facilitates that task [Terrasse, *et al.*, 2001]. These complex systems require demanding validation, refinement and model transformation into executable code. The automatic generation of code from models ensures that the specifications defined for the software solution in the design phase are implemented accordingly. The reutilization advantage of metamodelling is obvious, since several models can be derived from a single metamodel. Also, meta-metamodels are used as a synchronization basis for every conformed

metamodel, which constitutes a big advantage of metamodelling.

This dissertation's scenario or target domain is given by Primavera Business Software Solutions, S.A. (PBSS) concerning its ERP (Enterprise Resource Planning) [PBSS, 2007] software solution.

1.3. Research Goals

The goals of this dissertation are the following: (1) Use UML-based notation to conceive DSLs; (2) Conceive a metamodelling approach, based on the outlining of UML-inspired abstract and concrete syntaxes, to be considered during the definition of DSLs within Microsoft DSL Tools; (3) Build a demonstration case in order to reflect upon the undertaken approach in the context of a part of the Primavera ERP software solution (a portion of its sales domain).

This dissertation is dedicated to metamodelling as an approach to develop software in a more abstract way than purely programming the software solution. Domain Engineering is the field where it can be more accurately fitted in, with DSLs being its main artefact. Having UML-based DSLs is extremely important as it will be seen further on in this dissertation.

1.4. The Demonstration Case

The research approach used in this dissertation is the proof of concept, or concept implementation [Vessey, *et al.*, 2002]. The proof of concept research approach is about demonstrating the feasibility of a solution to a problem. The question with feasibility in this dissertation is whether it is possible to use DSLs based on UML notation, within DSL Tools, in the context mentioned above, thus, to a practical problem, or not [Shaw, 2001]. In order to demonstrate the feasibility of the proposed solution to the problem of using UML notation to build DSLs in DSL Tools, regarding a part of the sales domain of the Primavera ERP software solution, a series of mock-ups is used as a means of validation of the software development approach shown in this dissertation.

This section is intended to define the domain handled by the demonstration case presented in this dissertation. The focus is on the purchases, stock and current accounts modules of the Primavera Express[2] software solution. The main business objects are products, products families (merchandise, raw material, finished products…), customers, zones (store zones), sales people and suppliers.

[2] The version of Primavera Express used was 6.30.

Figure 1 shows some examples of business objects belonging to the Primavera ERP GUI (Graphical User Interface) in its free version.

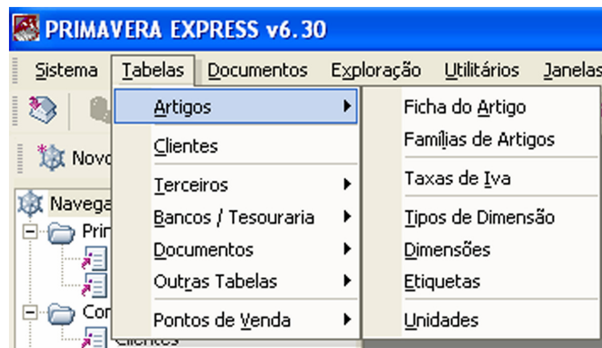

Figure 1 – Some business objects of the Primavera ERP

Following, the attributes of each one of the business objects: (1) Product (Identification, Description, Product family, Sale price, Purchase price, Supplier and VAT (Value Added Tax) percentage); (2) Product family (Identification and Description); (3) Customer (Identification, Name, Payment condition, Zone and Sales person (attributed to the customer)); (4) Zone (Identification and Description); (5) Supplier (Identification, Description and Payment condition); (6) Sales person (Identification and Description); (7) Payment condition (Identification, Description and Days for payment); (8) Purchase (Identification, Purchase number (automatic and sequential), Invoice number, Date, Supplier, State (ordered or invoiced), Payment condition, Purchased item and Total); (9) Purchased item (Identification, Product, Purchase price, Product description, Quantity, VAT percentage and Total).

The main interfaces available to the client of the ERP solution are: (1) Receive order from customer; (2) Issue invoice to customer; (3) Manage customer current account; (4) Send order to supplier; (5) Receive invoice from supplier; (6) Manage sales person current account; (7) Manage stock; (8) Receive payment from customer; (9) Pay order to supplier; (10) Consult products list, customers list; (11) Consult sales per product list, per customer, per zone, per sales person; (12) Consult orders per product list; (13) Consult available quantity in stock per product; (14) Consult list of movements for each stocked product (last inventory increment and last inventory decrement); (15) Consult customer debt list (per customer, per zone, per sales person); (16) Consult list of debts to suppliers.

Figure 2 contains some attributes of the business object *product* (*Artigo*), like sale price (*Pr. de venda em EUR*) and VAT percentage (*Taxa de IVA*).

Figure 2 – Example of the business object product

The demonstration case considers business objects' description and their attributes as the reference description when conceiving the structural model of the system, whereas the list of interfaces to become available to the ERP solution's client is considered as the basis for the conception of behavioural models and of the model of external functionalities and instances of the problem. It is not intended in this dissertation to optimize the logical architecture of the Primavera ERP's structural view [Machado, *et al.*, 2006].

1.5. Dissertation Roadmap

Chapter 2 of this dissertation presents an overview on the concepts crucial to realize DSLs in their full range. First, the MDD approach, the parent approach of the approaches MDA, SF, SPLs and DSLs, then metamodelling as an approach of MDD, followed by model transformation and software evolution, finishing with some software development methodologies and DSLs as part of the SF approach. Two of those software development methodologies are about SPL development, another one is about software development team composition and responsibilities and the remainder is about extracting logical architectures out of use case models.

Chapter 3 explores some ways of how metamodelling environments can be tools to handle DSLs, like Microsoft DSL Tools are. The role of the main participants in the development cycle of metamodelling environments is explained, along with the whole process of development of such environments. After that DSLs are viewed as languages that

can be metamodelled. The UML metamodel is partially presented due to its utilization in the definition of the DSLs in chapter 4. An approach to stereotype DSLs is presented at the end of the chapter. The approach can be used in modelling environments to apply domain-specific concepts to regular UML classes.

Finally, chapter 4 talks about Microsoft DSL Tools as metamodelling and design environments to conceive UML-based DSLs. In the first place, the initial experiment done with the tool is described, which includes code generation capabilities of it. After that, the main demonstration is reported. This experiment includes handling with metamodels in a metamodelling environment and handling with models in a modelling or design environment. A UML-based metamodel for the Primavera ERP sales domain is exposed and the whole process, as well as the resulted metamodel and respective models, are analysed.

2. Concepts to Understand DSLs

This second chapter of the dissertation gives the perspective of other authors on the topic covered by the document. The first major topic handled is concerned with MDD and metamodelling, the key approaches in which DSLs fit in. Secondly, model transformation and software evolution are explored as the main activities undertaken while modelling in general. Because of their extreme relevance, software development methodologies and DSLs are the last topics being mentioned in this chapter.

2.1. Introduction

First of all, an introduction to models in general will be exposed in this chapter of the dissertation. In order to understand the approach of DSLs, it is important that the intimate relationship between DSLs and other concepts, like software development methodologies, metamodelling, model transformation, software evolution and SF, is exposed as well.

The DSLs approach can be applied to the problem being handled in the development of a software solution through a metamodelling approach. The ultimate goal is to obtain the most efficient and effective implementation of that software application. Conceiving metamodels is a high abstraction way to accomplish this goal. The conceptual work involved in metamodelling DSLs is extremely domain-related. Nevertheless, a serious work in syntax and semantics is imperative and traverses all possible domains. The overall goal is to obtain

advantages in the whole software development process. Metamodelling and DSLs play a vital role in this aspect.

If a metamodel is conceived, transformations between models using automation tools are facilitated [Brown, *et al.*, 2005a]. Transformations between models are vital for a metamodelling approach involving DSLs. As Sendall and Kozaczynski [2003] stated, transformations are "the heart and soul of model-driven software development". Due to several scenarios, models need to evolve over the time of a software application development and transformations are part of that evolution.

This chapter of the dissertation presents some software development methodologies mentioned in the literature. The first two methodologies were chosen due to the fact that they were developed within the University of Minho and the other two because they are about SPL development. SPL development is part of the approach of SF (see section 2.4 of this chapter) along with DSLs.

2.2. MDD and Metamodelling

A model can be defined as the "collection of all the artefacts that describe the system" [Balmelli, *et al.*, 2006], or as an artefact "used in reasoning about the problem domain and the solution domain for some area of interest" [Brown, *et al.*, 2006].

Models are abstractions or representations of a system. They can be [Terrasse, *et al.*, 2001]: (1) Structural (class and collaboration diagrams); (2) Behavioural (state machine, activity and sequence diagrams); (3) Physical (component and deployment diagrams); (3) Of external functionalities and instances (use case and object diagrams).

In a similar way, Sprinkle, *et al.* [2001] described models as being artefacts capable of describing operations within a computer-based system and distinguish operations comprised of physical and others of information-handling processes.

According to Brown, *et al.* [2006], different models suit different purposes: (1) Establish a clear understanding of the problem; (2) Communicate a clear vision of the problem and its solution; (3) Generate low level implementations from high level models.

Models are developed both at problem and solution domains [Brown, *et al.*, 2005a]. The way from the problem domain to the solution domain demands for a deep knowledge in architecture [Cook, 2004]. To obtain the implementation of the logical architecture regarding the application in development, some abstraction levels must be traversed. Abstractions can be applied to procedures or data types [Bragança, 2003]. Each of the views or models on the

system captures the system's structure or behaviour with a specific abstraction deepness [Sendall and Kozaczynski, 2003]. Abstraction levels change according to the model's closeness to the target platform[3] [Metzger, 2005]. So, models can be more or less abstract, if they are closer or more distant from the implementation platform. Seifert and Beneken [2005] mentioned abstraction as zooming in or out of the models. Models can be organized horizontally and vertically, closer or more distant from the implementation level [Sendall and Kozaczynski, 2003]. Horizontally, models are attached to different views of the system. Vertically, models are attached to different levels of refinement or detail. Models must also be consistent with each other and the system's requirements. MDD is a Software Engineering approach that allows dealing, trough abstraction, with the complexity of developing a system and uses models to accomplish that. Hence, systems are modelled at different levels of abstraction. As the system's modelling evolves, transformations will be necessary to obtain models at different levels of abstraction. MDA distinguishes between models independent of their implementation platform and those that are not [Hailpern and Tarr, 2006]. The first ones are called PIMs (Platform-Independent Models) and the last ones are called PSMs (Platform-Specific Models) [Miller and Mukerji, 2003]. A PSM illustrates the way a PIM will be implemented in a particular platform. MDA tools are responsible for transforming PIMs into PSMs[4].

Recently the focus has been on system perspectives valuable to software engineers and software developers, and the generation of code as well. Perspectives on modelling are always around code, no code at all or some state in between [Brown*, et al.*, 2005a]. Brown*, et al.* [2005a] presented a division of those perspectives. In the code only perspective, it is difficult to manage the evolution of solutions due to scale and complexity issues. In the code visualization perspective, models suit the necessity to understand the code's structure and behaviour. The model is just another representation of the code. In the roundtrip engineering perspective, the generation of code from models is considered and the model is refreshed whenever the code is altered, yet, this is done without significant discipline. In the model-centric perspective, the implementation of systems is made from models. Here, the generation of code from models can be made through the application of patterns. In the model-only

[3] A platform is a set of subsystems and technologies to support the execution of a software application, as Atkinson and Kühne [2005] defined it.

[4] Besides PIMs and PSMs, MDA is also about CIMs (Computation Independent Models) [Miller and Mukerji, 2003]. CIMs model the system's requirements. They are independent of the system's implementation and are expected to show how the system will be used in its environment. They must gather a shared vocabulary on the problem domain to be used in other models. CIM's requirements must have corresponding constructs in the PIM and PSM that follow the CIM.

perspective, models in general are only an artefact to understand the problem or the solution domain.

As stated before in this dissertation, models are one of the essentials of MDD. The main competitive advantage MDD represents to organizations is productivity gain in the development of software. One of its main aspects is reducing the impact of changes on software artefacts. Table 1 describes the fundamental forms of change that Atkinson and Kühne [2003] presented.

Table 1 – Fundamental forms of change in software artefacts

Personnel	Software developers' knowledge on software artefacts should not be kept personal; software artefacts must be accessible to as most people as possible and be in a form understandable by every stakeholder.
Requirements	Online systems cannot be offline for long periods of time for the purpose of maintenance and, preferably, changes in software artefacts must be made at runtime.
Development platforms	Software artefacts should be disassociated from their development environments and be able to be exchanged between tools.
Deployment platforms	Platform-specific software artefacts must be obtained from platform-independent software artefacts.

Models allow efficiency and effectiveness of programming to be maintained throughout time, which means that any change in the code can be done with the least time consumption and the higher success probability. In other situations, models are necessary when establishing requirements with the customer, as well as obtaining certification in the software development area [Hailpern and Tarr, 2006]. Communication between developers and analysts may be highly sped up if models are used. Models may also function as a memory artefact useful when some developers leave the team and are replaced by others who need to keep up-to-date with what was developed before. The same happens with solutions that must be developed over long periods of time. Following this reasoning, the Software Engineering approach MDD makes sense. MDD imposes the structuring of the software development process around models adequate to each one of the moments within that process.

Atkinson and Kühne [2003] also defined the requirements for an MDD infrastructure. These are: (1) Availability and shared understanding of notation for creating models; (2) Rules for the use of models; (3) Understanding of the relationship between models and software artefacts; (5) Easiness of models interchange between tools; (6) Possibility to define mappings between models (other visual models or code).

With the goal of developing a set of standards built on modelling as the best practice to develop systems, the OMG presented the MDA standard [Mellor, *et al.*, 2004]. This standard includes a set of technologies (UML, profiles and others) and techniques that allow MDD to take place. Products and other standards which comply with the MDA standard are tagged with the MDA stamp.

Shortly, MDA is an approach of MDD. It equals semi-automatically generated code and use of standards for construction of models and transformations between them [Brown, *et al.*, 2005a]. Automation can be seen as executable patterns [Brown, *et al.*, 2005a]. Tools can generate code out of models that have been transformed into PSMs [Sendall and Kozaczynski, 2003]. MDA defines a process for creating models [Demir, 2006]. Instead of having to change features in source code to move between platforms, features can be changed in models and implemented in several target platforms as wished [Cook, 2004; Greenfield and Short, 2003]. That is the main idea of MDA: problem domain instead of technology domain.

MDA foundations are [Booch, *et al.*, 2004]: (1) Use of standards for its purposes; (2) Solutions intimately attached to problems; (3) Accurate designs; (4) Increased productivity; (5) Higher level of abstraction.

Visual modelling is one of the foundations of MDD. Other technological foundations are the support of OO (Object-Oriented) languages concepts and meta-level description techniques. The traditional OMG modelling infrastructure, or Four-Layer Architecture of UML, comprises a hierarchy of model levels, just in compliance with the foundations of MDD [Atkinson and Kühne, 2003]. Other authors like Demir [2006], Nordstrom, *et al.* [1999] and Lédeczi, *et al.* [2001] mentioned the Four-Layer Architecture of UML. Each model in the Four-Layer Architecture of UML, except for the one at the highest level, is an instance of the one at the higher level. The first one, user data, refers to the data manipulated by software. Models of user data are called user concepts models and are one level above the user data level. Models of user concepts models are UML concepts models. These are models of models and, so, are called metamodels. A metamodel is a model of a modelling language. It is also a model whose elements are types in another model. An example of a metamodel is the UML metamodel. It describes the structure of the different models that are part of it, the elements that are part of those models and their respective properties. The meta-metamodels are at the highest level of the modelling infrastructure, the MOF (Meta-Object Facility) [Alanen, *et al.*, 2005; Alanen and Porres, 2008; Jouault and Bézivin, 2006; OMG, 2007a]. The objects at the user concepts level are the model elements that represent objects residing at the user data level. At the user data level, data objects may be the representation of real-world items. UML

concepts define the language to express user concepts, as well as MOF defines the language to express UML concepts. Other languages at M3 level are ECore and KM3 [Alanen*, et al.*, 2005; Jouault and Bézivin, 2006].

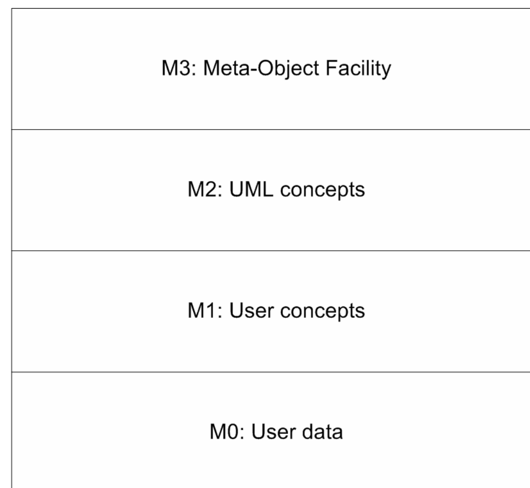

Figure 3 – OMG modelling infrastructure or Four-Layer Architecture of UML

Atkinson and Kühne [2003] identified two orthogonal dimensions of metamodelling (language definition and domain definition) and two associated forms of instantiation (linguistic instantiation and ontological instantiation, respectively). The levels of UML concepts and MOF are concerned with language definition. Linguistic instantiation takes place between elements at the user concepts level and elements at the UML concepts level and also between elements at the UML concepts level and elements at the MOF level. Ontological instantiation establishes an instantiation relationship between user concepts and also between UML concepts within the same level of the Four-Layer Architecture of UML. Atkinson and Kühne [2003] called ontological metamodelling to the description of domain concepts, particularly domain meta-types (or types of domain types). Ontological meta-types are distinguished from each other through stereotypes. So, ontological meta-types are used to distinguish the types themselves (a mammal from a reptile) but also to distinguish meta-type properties (different meta-types have different properties, thus a mammal uses a different locomotion means than a reptile does). The biological taxonomy for living beings or biological classification is an excellent example of ontological metamodelling. An MDD infrastructure must give equal relevance to both ontological and linguistic metamodelling strategies.

Regarding language definition Atkinson and Kühne [2003] divided the concept in four associated concepts: abstract syntax, concrete syntax, well-formedness and semantics. Abstract syntax is equivalent to metamodels. Concrete syntax is equivalent to UML notation. Well-formedness is equivalent to constraints on the abstract syntax (in OCL, or Object

Constraint Language [OMG, 2007a], for instance). Finally, semantics is the description of the meaning of a model in natural language.

DSDEs (Domain-Specific Design Environments) [Lédeczi*, et al.*, 2001] are tools that allow the capture of specifications through models conceived in a specific domain context. UML case tools are examples of these tools. Models conceived in a specific domain context are artefacts to design specific problem domains. From these models at the design level, implementation artefacts, like classes definition in a particular execution platform, can be generated using translators. DSDEs use DSLs (defined with the metamodels of the above mentioned models) to specify the system in its different views.

The Institute for Software Integrated Systems at Vanderbilt University conducted research on Model-Integrated Computing [Lédeczi*, et al.*, 2001]. This technology defines the DSL and model integrity constraints through metamodelling in order to compose the DSDE. Metamodels are made out of UML class diagrams and OCL constraints [Lédeczi*, et al.*, 2001]. A constraint can be, for example, prohibiting a transition of state to itself. Models syntax can be defined through metamodelling, as well as the relationships between model elements [Sendall and Kozaczynski, 2003]. The DSME (Domain-Specific Metamodelling Environment) [Nordstrom*, et al.*, 1999] must verify if the metamodel is semantically consistent (through OCL expressions, for example).

The Four-Layer Metamodelling Architecture of UML is very much appropriate to the context of DSLs [Lédeczi*, et al.*, 2001]. A metamodelling language, at the metamodelling level of the architecture (M2 layer), turns the possibility of specifying modelling languages suitable to various domains into a reality. A meta-metamodelling language can be used, at the level of meta-metamodels of the architecture, to specify the metamodelling languages themselves. The DSLs defined at the metamodel level allow specifying domain models for the software systems to be implemented.

GME (Generic Modelling Environment) [Zhang*, et al.*, 2005] is a metamodelling environment based on UML. It can be configured with specifications of a domain containing model elements and relationships of that specific domain to become a DSME.

DSMEs are metamodelling environments to specify DSLs which allow the generation of *domain-specific environments* to produce models that can be translated into implementation code or the application itself [Lédeczi*, et al.*, 2001; Sprinkle*, et al.*, 2001]. The environment where the metamodel resides can be specified through a meta-metamodel.

Metamodels require syntax and semantics specification. The metamodel has to

present all the entities and relationships among them to be handled with the intended language. Concepts like syntax and semantics are, thus, trivial [Atkinson and Kühne, 2003; Metzger, 2005; Nordstrom*, et al.*, 1999; Zhang*, et al.*, 2005]. The metamodel must express the obligatory semantic conditions for any model conforming with that metamodel, therefore, a valid model. The metamodel may include constraints like multiplicities and mandatory attributes.

2.3. Transformations and Software Evolution

Some general remarks on model transformation are given next, in Table 2 [Atkinson and Kühne, 2003; Brown*, et al.*, 2005a; Metzger, 2005; Schleicher and Westfechtel, 2001; Sendall and Kozaczynski, 2003; Terrasse*, et al.*, 2001]. Model transformation makes sense in the context of MDD. Code generation is the ultimate goal of the model transformation cycle. Thus, the mapping of models into code must be considered [Demir, 2006]. MDA refers to automated tools to transform DSLs into implementation code [Booch*, et al.*, 2004]. Concerning another OMG's creation, MOF QVT (Query/View/Transformation) [Dupe*, et al.*, 2007] is composed by a set of interrelated languages that suit the purpose of expressing transformations between models before they can be mapped into code. This OMG standard includes the language called QVT Operational Mappings [Dupe*, et al.*, 2007]. QVT Operational Mappings extends OCL. It establishes a compromise between the functional characteristics of OCL and the traditional constructs that may be found in Java. Recently, France Telecom R&D, partially financed by the European IST Modelware project, has launched the first open source tool to perform model transformations using the language QVT Operational Mappings – SmartQVT. It works as a plug-in of Eclipse.

Metzger [2005] classified transformations into endogenous and exogenous ones. An endogenous transformation takes place if the language of both the source and the target models is the same. An exogenous transformation, on the other hand, occurs if the language of the source model is not the same as the one of the target model. Brown*, et al.* [2005a] classified transformations into three possible transformation kinds. These are refactoring transformations (reorganization of a model), model-to-model transformations (entity classes to database schema, for example) and model-to-code transformations (code in a GPL, configuration files, deployment files, message schemas…).

Table 2 – Key concepts of model transformation

Notation	Transformation languages can use visual or textual notations. Sendall and Kozaczynski [2003] argued that transformations which use visual notations to define the input and output models are more appealing and cognitively easier to handle than others.
Programming style	Transformation languages can be declarative or imperative. Combining imperative and declarative approaches[5] was mentioned by Sendall and Kozaczynski [2003] as being an advantage for transformation languages.
Specialization level	Transformation languages can use general-purpose constructs or specialized constructs.
Abstraction deepness	Transformation languages must be abstract enough to be intuitive, thus, enhancing users' comprehension on the language.
Language range	Transformation languages must cover a wide range of transformation scenarios.
Automation	Transformation languages must provide automated model creation and maintenance.
Transformation composition	Transformation languages must provide the composition of transformations.
Customization	Environments to write transformations must make possible for users to define specialized transformations.
Checkability	It shall be possible to check a model for consistency with a given metamodel in order to perform code generation.
Automatic transformations	Automatic transformations can happen if both the source model's and the target model's syntax are defined.
Type of mapping	Transformations can be unambiguous and, so, provide support for a direct mapping. Transformations can require decisions regarding the use of alternative mappings. Transformations can be just an approximation if the target model doesn't match semantically with the source model.
Efficiency	Transformation efficiency is achieved by making transformations explicit and reusable.
Transformation rules	Transformations have rules that must be interpreted by compilers in order to generate code appropriately. A profile defines rules for model transformation.

As defined before in this dissertation, refactoring allows changing the internal structure and behaviour of software applications without changing its external behaviour. It

[5] If the declarative approach is considered, transformation rules include pre- and post-conditions. If the imperative, or operational, approach is considered, then, transformation rules are a sequence of actions.

works the same with the code as it does with the model [Zhang, *et al.*, 2005]. But the best strategy is to detect errors earlier in the development process instead of discovering them later on and needing to refactor the application code.

Transformations of models can be vertical, horizontal or oblique (a combination of both) [Greenfield and Short, 2003]. The refinement of transformations until code is generated is considered to be a sequence of vertical transformations. Refactoring (or improving without changing meaning) of the product's design means making horizontal transformations.

Transformations are useful when transforming views between different levels of abstraction, but they are useful as well when transforming models at the same level of abstraction [Brown, *et al.*, 2005a]. Thus, models can be refined, which is the same as adding details to them [Brown, *et al.*, 2005a].

In the process of mapping a model (or more than one model) into another model (or more than one model), a mapping function is involved. This function defines the mapping rules that allow the transformations between source and target model or models to occur. The main characteristics of model mapping are creation and synchronization. Mappings are well suited to create models with other models (model derivation). This way, synchronization between models is assured. Mapping functions represent repeated design decisions which conduct to the reuse of those functions in models of similar design.

The transformation of an analysis model (model at the problem domain level) into a design model (model at the solution domain level) is made by means of mapping functions. Having EJB (Enterprise JavaBeans) [Mellor, *et al.*, 2004] as a reference, two kinds of objects or beans may be taken into account: those whose lifetime is the same as the time period in which they reside in a database (entity beans) and those whose lifetime corresponds to the session lifetime at the client side of the application (session bean). The model of a bank at the analysis level comprises elements like its clients and their accounts. The model of a transfer at the analysis level comprises elements like amount and involved accounts. At the design level the bank corresponds to an entity bean and the transfer to a session bean [Mellor, *et al.*, 2004]. A mapping function is responsible for the transformation of the two models at the analysis level into the corresponding types of beans at the design level.

Atkinson and Kühne [2005] defended several transformations between a PIM and a PSM instead of a unique transformation. That is why a model transformation may give birth to either other models that still need to be transformed or different levels of generated code [Brown, *et al.*, 2005a]. When a DSL model reaches an abstraction level that allows its direct

transformation from model to source code (with no more models as intermediates in the transformation process), then the DSL elements are mapped into constructs of a target platform. Template languages allow this direct transformation to take place [Bettin, 2004].

Software maintenance, at a high level, may be due to four kinds of reasons as pointed out by Seifert and Beneken [2005]: (1) Due to changes in the requirements with impact on the implementation of functionalities – it is called perfective maintenance; (2) Due to technological constraints from the exterior (runtime environment, external components, development tools…) – it is called adaptive maintenance; (3) Due to errors that must be corrected – it is called corrective maintenance; (4) Due to the avoidance of other kinds of maintenance – it is called preventive maintenance.

Software evolution takes place, at a lower level, when: (1) Changes need to be performed in the syntax and semantics of existing domain models [Sprinkle and Karsai, 2004]; (2) Software needs to be rewritten due to novel requirements [Batory, *et al.*, 2002; Bettin, 2004]; (3) The DSL changes [Batory, *et al.*, 2002]; (4) Changes take place in the DSVL metamodel (newer versions of a DSVL that prompt the existence of the concept of domain model evolution at the metamodelling level) [Sprinkle and Karsai, 2004]; (5) Changes in the architecture [Seifert and Beneken, 2005]. These changes may oblige transformations to take place.

Some Software Engineering solutions that cover software evolution are [Batory, *et al.*, 2002]:

(1) *Object-oriented design patterns*; Design patterns are an approach for redesigning and generalizing design traces of object-oriented software. Software evolution in this case occurs when a design pattern is applied to an already existing software solution design.

(2) *Product line architectures*; Product line architectures represent reusable designs of product line members. Software evolution occurs when components with new features implemented are added to the overall architecture or others are taken out of the same architecture.

(3) *DSLs*; Finally, when the software needs to evolve, DSLs must meet the required changes in software caused by that evolution.

2.4. Software Development Methodologies and DSLs

This section of the second chapter is focused on software development methodologies. Some methodologies are presented: 4SRS, VA, FAST and SF (SF include

DSLs). The first two were developed within the University of Minho and the others are concerned with SPL development. Modelling without the guidance of a methodology can be ineffective and is, in general, costly, which explains the significance of software development methodologies within this context. The VA methodology is about the roles of engineering professionals within the computer-based systems development. FAST and SF methodologies are much about the distinction between the commonalities and the variabilities of SPLs. Further on in this dissertation there will be some reasoning around roles of engineering professionals partially inspired by the VA methodology, as well as it will be shown a metamodelling approach targeted at a SPL, the ERP SPL, which distinguishes between the common aspects of the SPL and the aspects which vary across the SPL's products. 4SRS is considered to be an appropriate methodology to be used after a first UML-based metamodelling exercise (like the one this dissertation describes) in order to avoid ad-hoc metamodelling.

2.4.1. 4SRS

The first methodology is called 4SRS (Four Step Rule Set) [Machado, *et al.*, 2006][6]. Its goal is to provide a method to transform requirements specification into software logical architectures by means of recursive model-based transformations, basically, from UML use case diagrams to UML 1.x object diagrams or UML 2.0 component diagrams. 4SRS is a solution for the problems that emerge from ad-hoc modelling principles during the design phase of a software project, specifically the architectural design of the system. Identifying and relating architectural elements which are part of a system is crucial to the success of its design and further implementation [Machado, *et al.*, 2006].

Functions provided by application components can be called services. Standard descriptions of those services make it possible to compose the structure and behaviour of application components. The software engineer must specify the interactions of the system with its users in the form of a use case diagram (also called functional or analysis model) and from there, following a series of steps, get to the architectural components of the system (also called conceptual or design model). 4SRS decisions on how to transform functional into structural models is not intuitive but rather methodical [Machado, *et al.*, 2006]. That is the value of the method.

The definition of system's objects is made out of the textual descriptions of system's use cases, which reveals the significance of Requirements Engineering in the whole process.

[6] This technique was developed within the University of Minho, supported by projects STACOS and USE-ME.GOV.

In the process of converting a use case diagram into a component diagram we have to separate objects into types (interface, data and control) and establish a relation between the object and the use case from which it was born. Ultimately, the behaviour of the system is specified out of its functionalities.

Four steps constitute the method 4SRS: object creation, object elimination, object packaging and aggregation, and, finally, object association. The object elimination step is the crucial one, as it allows to remove redundant requirements and to find missing requirements. When it comes to object packaging and aggregation the goal is to semantically group objects, hence, it consists of a task situated in a high abstraction level. Packages define regions whose architectural structure must be further specified in the design phase of the project by means of design patterns. Raw component diagrams can be collapsed or filtered in order to define smaller projects within the overall project. Therefore, the boundaries of the system can be defined and redefined using raw component diagrams' collapsing and filtering. The raw object model represents the service architecture and from it class diagrams, state machine diagrams, activity diagrams and sequence diagrams can be drawn [Fernandes, *et al.*, 2006].

2.4.2. VA

The second methodology is called VA (Virtual Automation) [Machado and Fernandes, 2002]. IIS (Industrial Information System) can be viewed in terms of the implementation project's team structure, as well as in terms of associated activities, using the VA methodology. This methodology mentions roles like the one of the hardware engineer, the one of the software engineer and the one of the information systems engineer.

The VA methodology is divided into levels. Level 1 is the one where the hardware engineer is contextualized. At this level, the algorithms for the IIS must be implemented in terms of hardware and made transparent to level 2. This level 2 of the VA methodology is where the software engineer performs his activities, which are to design the FMOTS (Functional Modules Off-The-Shelf), commonly named components, with a CASE (Computer-Aided Software Engineering) tool. Finally, level 3 of the VA methodology is concerned with the activities undertaken by the information systems engineer using a CAE (Computer-Aided Engineering) tool.

The advantages of the VA methodology are: (1) Reduction of development times; (2) Consistent design processes with reuse of activity outputs from previous software projects; (3) Effective design of systems at the Information Systems Engineering level accomplished through the integration of tools at different levels.

2.4.3. FAST

The third methodology is, in fact, a product line development process and is called FAST (Family-Oriented Abstraction, Specification and Translation) [Ardis, *et al.*, 2000][7]. It employs a process called commonality analysis in which the development of an application engineering environment is based on. This environment is then used to produce family members as fast and as cheap as possible.

The telecommunications operator Lucent claims to have successfully employed the FAST process in various projects with efficiency improvement and an increased consistency in products. This claim is perfectly reasonable since the telecommunications sector is one of large systems built upon smaller subsystems with particular characteristics. When Lucent began to use Domain Engineering, the productivity of developers increased and the quality of the code was improved. The main advantages in the use of FAST were: (1) Reduction of development times; (2) Increased behaviour consistency among all products; (3) Reduction in maintenance costs; (4) Reduction in training costs to have new employees familiarized with the new domain.

FAST assumes it is more advantageous to have only one instance of each common feature (that is the main premise of this methodology). FAST is divided into a phase of domain engineering and a phase of application engineering. The first one is about understanding how commonality and variability coexist and document that knowledge. The second one is about translating it into technology, like a set of subroutines (or component) or a DSL, the above mentioned application engineering environment. Any change to the environment must be done after checking its impact on the gathered knowledge. This is done in the context of the domain engineering phase. Also in the domain engineering phase, commonality analysis is a domain analysis method of FAST. This method is about the documentation of commonalities and variabilities of the product family. The commonality analysis documentation is essential when communication between marketing professionals, senior managers and developers is concerned.

A prototype can be used as a simplified version of an individual product. The prototype functions as a demonstration of especially the variability among the different product family members.

[7] FODA is another product line development process that concentrates in the selection and analysis of features to produce individual product members.

2.4.4. SF and DSLs

SF may be, in fact, considered another software development methodology. Some requirements characterize the success of the approach [Greenfield and Short, 2003]. Problem domain knowledge is extremely significant. Custom component suppliers emerge as specialists in several domains. In technical terms, the use of architectures requiring the capability to adopt components perfectly matching each other is crucial to the success of SF. Requirements Engineering has in SF extreme relevance through requirements capture and analysis.

When a family of software products shares the same features (common features, like behaviour, interfaces or code) it can be called a SPL, although the product line development process must comprise not only the concern with the commonality of products but the variation among them as well. This approach of SPLs is contextualized in the Domain Engineering area. One of the main goals of SPLs is to increase the generation speed of individual products [Ardis, *et al.*, 2000]. The maintenance and production of upcoming versions of a product family is very much reduced if SPL development is taken into account. Lucent is a case of success in this [Ardis, *et al.*, 2000].

SPLs are software systems that share common features considered as necessary for a market segment [Frankel, 2005]. A product family is composed of products. The development of family members is done using patterns, frameworks (a framework can be seen as all the code that implements common domain aspects and extension points to customize applications built from that framework; the model must specify the way to complete those extension points [Cook, 2004]), models and tools. SPLs are all about the distinction between commonalities and variabilities [Ardis, *et al.*, 2000]. Within the software development process of SPLs two processes are related to each other [Frankel, 2005]: the one that designs an architecture for the SPL framework, which is the core asset development, and the other that uses the framework to produce individual products, which is the product development.

The product line scope can be defined through a DSL [Frankel, 2005]. Variabilities were defined by Mernik, *et al.* [2005] as the information required to instantiate a system out of a broader one. Variabilities may be defined with a DSL [Frankel, 2005; Mernik, *et al.*, 2005]. Variabilities can be encapsulated in design patterns [Ardis, *et al.*, 2000]. Despite this, a generic architecture detaining commonalities between product line members must also be defined [Roubtsova and Roubtsov, 2004].

The major benefit of using a DSL is the reuse power attached to it. DSLs make available for reutilization a set of software artefacts like, as Mernik*, et al.* [2005] mentioned, language grammars, source code, software designs and domain abstractions. DSLs permit the reutilization of software architectures faced as the design the application generator follows when compiling the DSL. Also, with an API (Application Programming Interface) the reutilization of source code is granted. In fact, a DSL can be a GPL combined with an API, which is no more than a domain-specific vocabulary [Bragança, 2003; Mernik*, et al.*, 2005]. Applications can be built with a GPL compiler using a GPL combined with an API [Atkinson and Kühne, 2005]. The specificity of DSLs has to be considered [Bragança, 2003]: a DSL may have on its own all the abstractions necessary to specify the domain instead of using a library. In the case of the DSL being able of providing the means to construct new abstractions besides the ones it allows by default, then, the DSL is no longer a DSL and turns into a GPL.

Greenfield and Short [2003] identified three types of DSLs, among others, and those were: (1) Business entity DSLs (examples of business entities are customer and order); (2) Business process DSLs (examples of business processes are submission of order and calculation of discount); (3) Web service DSL (description of how business entities and business processes are implemented as web services in a service-oriented architecture).

Abstraction is concerned with emphasizing some characteristics of a system relevant to some stakeholders or purposes instead of other characteristics. Granularity is related to the size of software constructs. Specificity is related to the scope of the abstraction. According to the Three Axes of Critical Innovation, for the SF development, presented by Greenfield and Short [2003], abstraction values range from concrete to abstract, granularity values range from fine grain to coarse grain and, finally, specificity values range from single use to reusable. Abstract solutions are presented as requirements, whereas concrete solutions are presented as executable ones. Fine grain solutions are presented as lines of code, whereas coarse grain solutions are presented as Web services, for example. GPLs are of single use, whereas DSLs are reusable. Coarse grain software constructs are of more independent handling (during analysis, conception, implementation…), since the abstraction level is high. The more specific the scope of an abstraction is, the less granted the potential of reutilization of that software construct will be. Bragança [2003] described DSL, in the context of programming language classification, as being a vertical language or an application specific language and, so, the DSL can only be applied to an application or family of applications, thus, to a knowledge domain only.

The development of a DSL asks for domain and language development expertise [Cook, 2004; Mernik*, et al.*, 2005]. The DSL's design must be created by highly skilled software engineers with experience on the domain due to the complexity of such a task [Demir, 2006]. After the DSL is conceived, requirements engineers, domain experts, professionals without much programming skills and even the customer can understand it. The ontology set (set of domain concepts) of a DSL makes of the DSL an easy to learn programming language, compared to GPLs [Sprinkle and Karsai, 2004]. The conscience of the potential a DSL has may only be possible when already a lot of knowledge around the development of domain-specific software has been done using a GPL [Mernik*, et al.*, 2005]. Only then it is likely to realize the value of software reengineering or software evolution using a DSL. The main reasons for disregarding the implementation of DSLs in organizations are lack of expertise but most of all inexistence of short-term benefits, particularly when the question is to develop a new DSL, but also when the question is to find an existing DSL to suit particular needs (DSL documentation is commonly not well spread out of the organizations). To help organizations decide on whether or not to implement a DSL, Mernik*, et al.* [Mernik*, et al.*, 2005] presented a decision pattern. As Table 3 presents, different strategies can be adopted when designing DSLs [Mernik*, et al.*, 2005].

There is a close relation between DSLs and software development methodologies, like FAST and FODA (Feature-Oriented Domain Analysis) [Mernik*, et al.*, 2005]. The analysis phase in the development of a DSL is pretty much similar to the one of FAST. In fact, FAST or FODA can be used in this context. The problem domain must be defined, the domain knowledge must be gathered (technical documentation, knowledge from domain experts, pre-existing GPL code, customer surveys). After analyzing all the domain knowledge, domain-specific syntax and semantics must be determined.

Mernik*, et al.* [2005] mentioned, besides the definition of the domain scope and the domain terminology (same as overview and technical terms from FAST) as the output from the analysis phase, feature models to describe the commonalities and variabilities of domain concepts (same as structured lists from FAST) and the relationships between them. Various DSLs can be developed from a single feature model. FODA works with feature models.

Table 3 – Strategies to design DSLs

Designing a DSL based on an existing language	The easiest strategy to adopt. Its advantages are easiness of implementation and easiness of utilization (the easiness of utilization is only a fact if users already handle the existing language).
Designing a DSL by extending an existing language	This strategy may raise issues of integration between existing language and DSL.
Designing a brand new DSL	This strategy involves a difficulty to the DSL's analyst, since the DSL is going to be used not only by programmers but also by professionals with no or almost none knowledge in a programming language.

SPLs are mainly concerned with domain (which is a specific area of knowledge), hence, making use of DSLs understandable by domain experts [Bettin, 2004; Demir, 2006]. In the context of SPLs, if a DSL is defined during the analysis phase, then this language must be able to let users determine the values of variabilities between family members [Ardis, *et al.*, 2000]. The specifications in that language produced from the determination of variabilities values are then translated into code. The strategy to deal with a DSL in this context is to model the commonalities and then specifying the variabilities with a DSL [Ardis, *et al.*, 2000]. The DSL describes the concepts the framework presents. Variability points in a domain-specific framework may be filled by using a domain-specific model. This is called framework completion [Greenfield and Short, 2003]. Developers then need to write small amounts of code in a DSL to complete a framework in order to customize products to meet specific requirements.

Patterns have a marked presence in the DSL field [Mernik, *et al.*, 2005]. Analysis patterns are used in the context of domain analysis, design patterns in the context of DSL design and implementation patterns in the context of DSL implementation. These three kinds of patterns are independent but patterns inside each kind may be overlapped. A pattern, when applied to a model, changes its elements (adds new elements or properties to the model) in order to conform to the pattern directives [Brown, *et al.*, 2005a]. A pattern is a model with holes that can be fulfilled with another model or part of it [Cook, 2004].

DSLs are defined through metamodels. The more specific the metamodel is, the more specific the model conceived in a specific domain context can be [Sprinkle, *et al.*, 2001]. Terrasse, *et al.* [2001] mentioned the trend to define specific metamodels for each application domain from where application models can be derived. Domain-specific model elements can be added to UML as an extension of it in order to make UML applicable to specific domains. Thus, a DSL metamodel can describe, in fact, an extension to UML for a specific domain (a

correspondence between the UML metamodel elements and the DSL metamodel elements must be perfectly possible). An approach to extend UML is by using stereotypes, tagged values and constraints[8], expressed in OCL, for example. Stereotypes can be restrictive or constrained [Schleicher and Westfechtel, 2001]. A constraint on a stereotype can e.g. state that a pair of classes stereotyped with that particular stereotype can only be associated by a single instance of an association with a specific stereotype. Profiles can be considered as DSLs with stereotypes.

Demir [2006] used a PDM (Problem Domain Matrix), reporting to the analysis phase, and a SDM (Solution Domain Matrix), derived from the first and reporting to the design phase, to extract the commonalities and variabilities out of the product line and after that derive the DSL.

A DSVL is a DSL with a visual programming interface. Even a professional who is not a programmer can manipulate the DSVL as an application development tool (that is the main advantage of it). In this context, Sprinkle and Karsai [2004] distinguished between abstract and concrete syntax. They assumed that the abstract syntax available to build a DSL is designed with a metamodelling language through UML class diagrams and that the concrete syntax comes from a mapping between the elements in the abstract syntax and the visual constructs of the DSVL. The semantics of the DSVL must also be specified. Nokia uses a DSVL instead of UML to develop mobile phone software.

There is no agreement on whether a DSL must be executable [Mernik, *et al.*, 2005]. DSLs range from nonexecutable ones to fully executable ones. These last ones have an associated application generator which functions as a compiler of the DSL. During the creation of the DSL, the constructs of the GPL to be used when generating the application's code must be taken into account so that the code generator doesn't demand in runtime items not contemplated in the DSL metamodel [Bragança, 2003; Demir, 2006]. It is possible to create domain-specific embedded languages, whose (domain-specific) constructs are implemented through GPL constructs. This way the debugging task in a GPL compiler is not an issue but the design of the DSL is limited to extendable GPL constructs [Bragança, 2003].

Bragança [2003] summarized the advantages of using DSLs: (1) Quantity of code is reduced (as well as errors in code); (2) Productivity is increased; (3) Maintenance costs are decreased; (4) Closeness to problem domain; (5) Manipulation by domain experts is easier; (6) More understandable programs; (7) End users can participate in programs evolution;

[8] Structural constraints can be defined through class diagrams, a familiar way for a modeller to work with constraints.

(8) Documentation is facilitated; (9) Domain knowledge lifecycle is extended; (10) Testing becomes easier; (11) Programs can be validated and optimized before being executed from models; (12) Increased portability; (13) If applied to a SPL, costs of developing a DSL are reduced.

2.5. Conclusions

Models are the fundamental piece of Software Engineering and are the basis of several DSLs. Diverse advantages are available when using models to build software solutions, like smoothing the hard task of handling complex systems, turning team coordination an easier task, softening the impact of changes during the development process, increasing productivity and increasing the easiness of bringing the customer to the analysis phase of the software development project. It is inevitable to talk about code generation from models these days, so, models should provide also the support for the generation of code. Transforming models into code is a task of high value in the whole process of developing a software application with a DSL. Here, the relation between DSL and GPL plays a significant role when moving from models to implementation. MDD is at the centre of this question and brings up model transformation to light. DSLs are a part of the MDD approach called SF. Models are nothing without the framing of a methodology, like SF. In the context of SF, DSLs must make available the opportunity to determine the variabilities of a SPL in order to realize it in individual products. Model transformation is the way to produce those individual products out of models. MDA is another MDD approach with the advantage of being a standard and making use of standards. The main concern of MDA is precisely with model transformation (from PIMs to PSMs). But it is not sufficient to run transformations just one time. Software maintenance is a very important activity when working with models to generate software solutions over time. DSLs are an approach that covers software evolution with a specific kind of solution. Every time a DSL changes it means that software solutions must evolve from their previous state to a new one defined by the DSL metamodel.

To conceive a domain-specific modelling language it is of extreme relevance to be aware of metamodelling concepts, the fundament of every modelling language. The main task of a software engineer is to think about models and adopt a scientific reasoning to be used during the conception of a domain-specific modelling language. Using mainly abstraction to metamodel, software engineers should consider the enormous power of reutilization a metamodel has and the equally enormous advantages of that. The concern should be not only with metamodels but with the definition of the metamodelling language as well. All of this

thinking must be scientific, otherwise, metamodels will fail and DSLs will be chaotic and a complete failure. Software engineers, when metamodelling, should be worried with some very important concepts, like syntax and semantics.

Despite the reluctance of large organizations in adopting and some disadvantages of their use, DSLs convey various advantages to the development of software applications compared to GPLs, although in order to implement DSLs, GPL constructs may be considered and used. In fact, DSLs may be faced as a strategy of GPLs' adaptation to Domain Engineering.

As it was exposed in this chapter of the dissertation, DSLs are nothing if they are not conceived within a metamodelling reasoning framework. If software producers decide to use models in the development of their solutions, they also have to decide that metamodelling, besides model transformation, must be a part of that decision, otherwise, using models to produce software solutions is going to be worthless.

3. Metamodelling and Modelling Environments as Tools to Handle DSLs

The third chapter of the dissertation is targeted at exploring metamodelling environments as tools to conceive DSLs. The development cycle of metamodelling environments is firstly presented as a way of exposing the three main professional roles mentioned throughout the rest of the dissertation and dividing the conception of DSLs between the three associated stages also defined in the cycle. Then, a metamodelling perspective on DSLs is presented by bringing up the equivalence between DSLs and stereotyped class diagrams, followed by a short summary of the UML metamodel. The last section of this chapter is about stereotyping applied to the context of this dissertation: a DSL for the sales domain of Primavera ERP software solution (defined within a metamodelling environment).

3.1. Introduction

Metamodelling environments can be modelled themselves. In this chapter of the dissertation what can be called the development cycle of metamodelling environments is exposed. It comprises the roles of three professionals: domain engineer, software engineer and software developer. The cycle is divided into levels according to the Four-Layer Architecture of UML. Objects resulting from each level are also reported. Usually, the responsibilities of these professionals are not the appropriate ones and, so, this cycle is a way of following a rule. The final output of the whole process of metamodelling environments'

development is a metamodelling environment where metamodels can be conceived. Models are present throughout the whole way.

Along with the process of developing a metamodelling environment, a series of transformations may occur. These transformations may occur at the same level of abstraction as well as between different levels of abstraction.

DSLs can be seen as stereotyped metamodels, as it will be further explained in this chapter. But most of all, before metamodelling it is extremely pertinent to follow the basic concepts of the UML metamodel (its syntax and semantics). Some parts of the UML metamodel, which are presented at the end of this chapter, are going to be used until the end of next chapter of this dissertation. Also, a stereotyping approach is shown as a way of configuring instances of UML concepts (e.g. class) with domain-specific concepts in modelling environments.

3.2. Development Cycle of Metamodelling Environments

Figure 4 depicts the development cycle of metamodelling environments. The model specifies this development process, which is the process of developing a specific software application (final solution) for metamodelling software applications in general. The final solution is a newer version of the meta-design environment (same as metamodelling environment). But the final solution can be other than a meta-design environment. It can be another kind of software application composed of code or FMOTS (from VA). In that case, the model would have another name, development cycle of software applications out of metamodelling environments. The process points specific responsibilities to the project team members. It begins with the software developer at the implementation level, then moves to domain engineer at the analysis level, after that to the software engineer at the design level and finishes with the software developer at the implementation level. The software developer is responsible for implementing the meta-design environment from which the final solution is going to be implemented by him as well at the end of the process. The relation between each project phase is iterative and depends on the predecessor's ability to answer to all open issues. The objects resulting from each creation action have an equivalency to the levels of the Four-Layer Architecture of UML. Therefore, DSMEs can be produced from metamodels [Nordstrom, *et al.*, 1999].

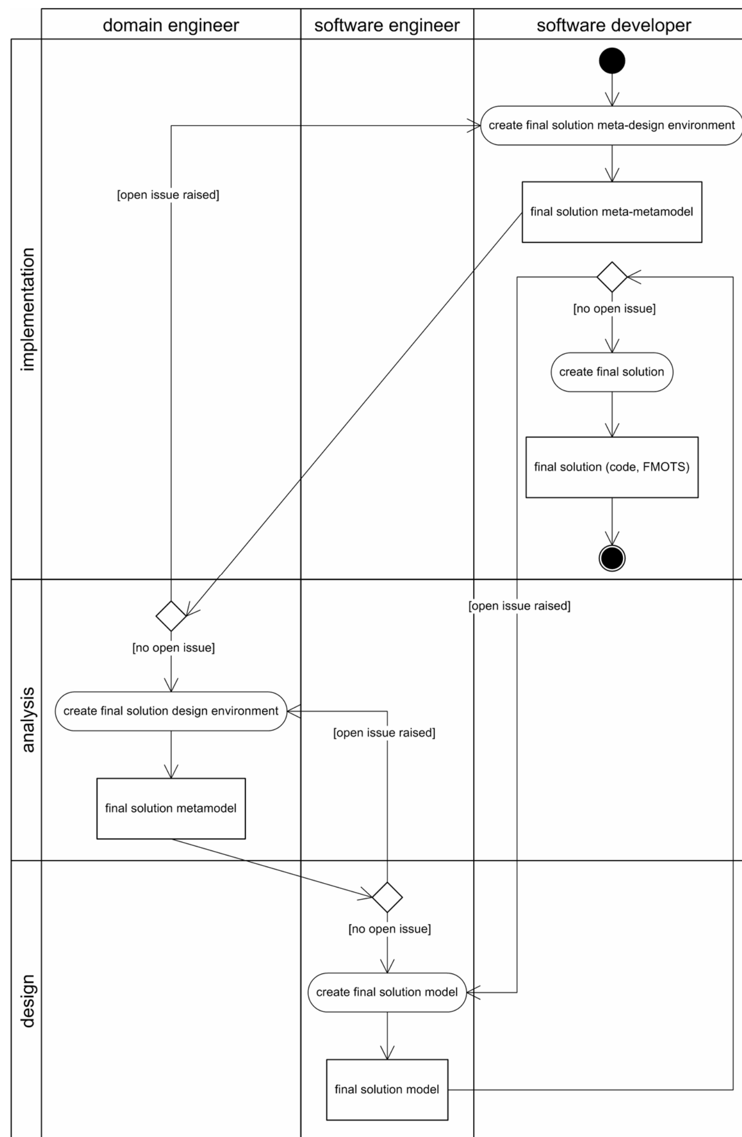

Figure 4 – Development cycle of metamodelling environments

The great goal of DSLs is accelerating the process of software development. They allow the possibility of having a specification language, of high abstraction level, capable of being manipulated by software engineers (the DSL) distinct from an implementation language, of low abstraction level, capable of being manipulated by software developers (the GPL of which the generated code out of the models designed with the DSL is made of). The software engineer ends up by having an artefact available that allows him a bigger understanding of the software to implement, whereas the software developer ends up by inserting less code manually, obtaining gains in productivity and the possibility of concentrating in the optimization of the business logic to implement. The domain engineer is the professional who is responsible for studying the problem domain (analysis). The software developer is the professional who is responsible for implementing the solution to the problem delimited by the domain engineer. Finally, the software engineer is the professional in the

middle of these two, responsible for studying the solution domain (design) which is going to be the input for the software developer's job.

When handling DSLs to develop metamodelling environments it is relevant to consider DSL transformations. Table 4 shows the types of DSL transformations that may occur between what can be called static DSLs and what can be called dynamic DSLs. The types depend on the level of abstraction the DSLs are at (if they are at the same or at different levels of abstraction). Static DSLs model structural aspects of the system to be specified, like class diagrams. Dynamic DSLs embrace all nonstructural aspects of the system, like use case diagrams and component diagrams.

Table 4 – Types of DSL transformations considering abstraction as a variable

Same level of abstraction	Static DSLs to Static DSLs: Refactoring or Intra-view Transformation.
	Static DSLs to Dynamic DSLs: Inter-view Transformation.
Different levels of abstraction	Static DSLs to Static DSLs: Refinement Transformation.
	Dynamic DSLs to Dynamic DSLs: Refinement Transformation.

3.3. DSLs from a Metamodelling Perspective

Conceiving DSLs is equivalent to stereotyping. According to the perspective of Schleicher and Westfechtel [2001], and following the Figure 5, extending the UML metamodel to define a new DSL means instantiating MOF itself, since the UML metamodel is already an instance of MOF. Using a stereotype is the same as instantiating a new UML metaclass, a special kind of UML metaclass which extends predefined UML metaclasses. If both DSL and stereotyped class diagram are instances of the UML metamodel, then, they are equivalent, which means that stereotyping is, in fact, specifying a DSL. Most of all, this approach takes advantage of UML, an already well-known standard among software industry [Booch, *et al.*, 2004].

It is from the files that define the language syntax that the more general characteristics of the domain model or model are translated automatically into code. The code contains the definition, in a GPL, of object types specific to a domain, or domain-specific data types. These object types, like page or screen, for example, are designed with the DSL. DSLs also contemplate procedures specific to a domain, or domain-specific procedures.

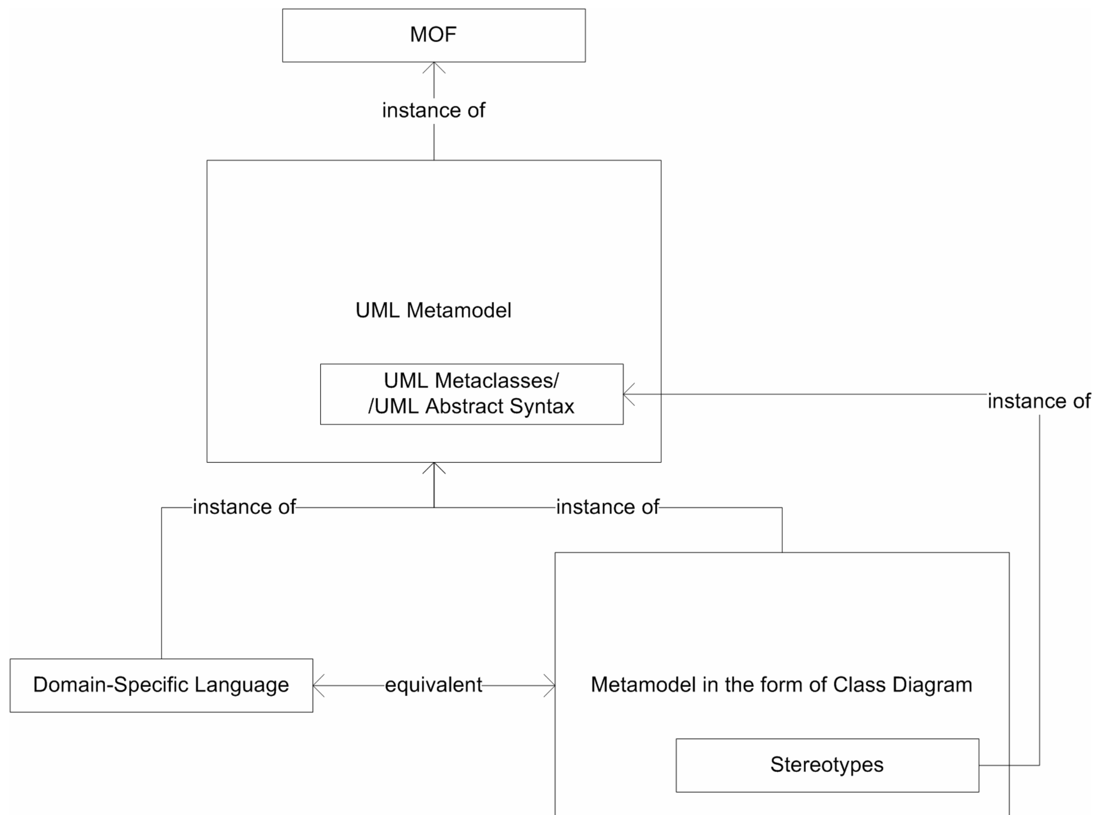

Figure 5 – Equivalence between DSLs and stereotypes

It is possible to refine the primarily generated code with code inserted manually by the software engineer, eventually also calling already created components by software developers, giving birth to a framework more consistent between new code and reused code.

The specification of the UML superstructure, or the UML metamodel's syntax and semantics, is now going to be described, in particular, the notation and meaning of some views (use case diagram, class diagram, activity diagram and state machine diagram) [OMG, 2007b]. These concepts must be kept in mind when metamodelling with DSLs respecting the UML metamodel.

The first view is the use case diagram. The concepts used to model use cases are shown in Figure 6. A use case diagram has the purpose of specifying the uses of the system. Usually, they are used to retain the system's requirements in terms of functionality, meaning what the system must be able to do. Each use case diagram has a subject, which is the system being specified. Actors are also represented in the diagram and refer to other systems, or even subsystems of another system, which interact with the system being modelled and are situated outside this system. The subject is specified with one or more use cases associated with specific needs of each actor, so, it makes sense to have every use case connected to at least one actor. In terms of semantics, an actor models an entity external to the subject. In an interaction between an external entity and the subject, that entity plays the role of actor. A use

case represents a particular behaviour of the system. Two use cases cannot be associated (yet, they can be connected through a relationship) because both, by themselves, describe a complete use of the system. The internal behaviour of the actor is not described in this diagram, only its interaction with one or more uses of the system is. As to the extend relationship, represented in Figure 6 by the *Extend* class, an extending use case is a use case inserted in the description or executed in one place or more places of the extended use case's description or execution. The extending use case and the extended use case are fused when the execution of the last one takes place. The extending use case is an increment to the extended use case. The extended use case just invokes the extending use case and is not aware of its behaviour. An include relationship, represented in Figure 6 by the *Include* class, between two use cases means that the behaviour defined in the included use case is included in the behaviour of the including use case. This kind of relationship is used when there are parts in common between two or more use cases. Those parts in common are extracted from the various including use cases to a separate use case (included use case) to be included by the various including use cases. The behaviour of the included use case is know by and described in the including use case.

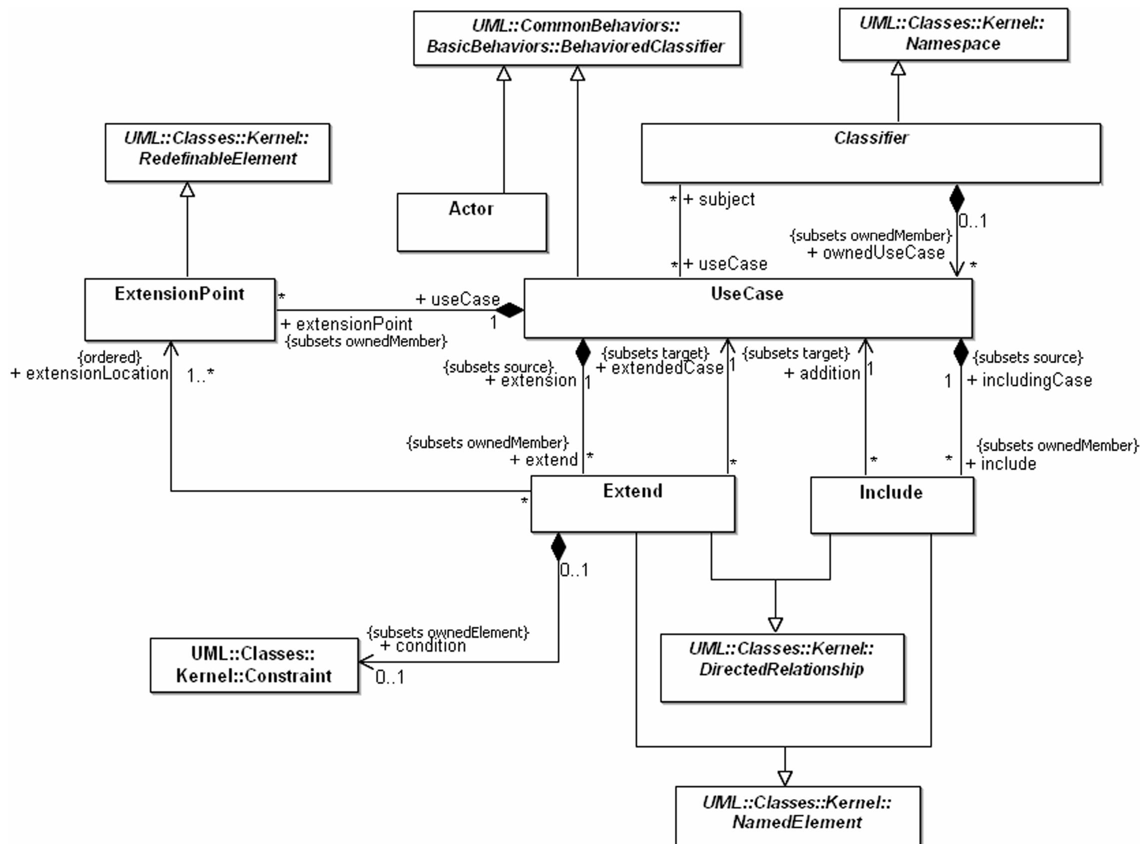

Figure 6 – Representation of the concepts actor, use case, extend and include used to model use cases (from the UML superstructure)

The class diagram has the purpose of describing the structure of the system being specified. A class, represented in Figure 7 by the *Class* class, has the purpose of specifying the object's classification, its (structural) characteristics and associated behaviour.

Figure 7 – Representation of the concept class used to model classes (from the UML superstructure)

A class can be represented by three compartments: a compartment with the name of the class, another compartment with the list of its attributes and another with the list of operations associated with the class. An association, represented in Figure 7 by the *Association* class, states that instances of associated types are connected. It can be bidirectional or unidirectional. The values at each association's extremes indicate the number of instances of the associated type connected to that extreme of the association. It is possible to have several associations between the same pair of associated types. When that happens, not only the association is tagged with the values at each association's extremes, but also with an additional identifier at each of those extremes. An association can represent as well an aggregation or a composition. A composition is a strong form of aggregation. If a composition is erased, all the parts that are related through the composition are erased as well.

An association may have its own attributes. That is the case of an association class (an association and a class, simultaneously), represented in Figure 8 by the *AssociationClass* class. The extremes of associations between association classes and classes from the association class' side have always multiplicity equal to 1.

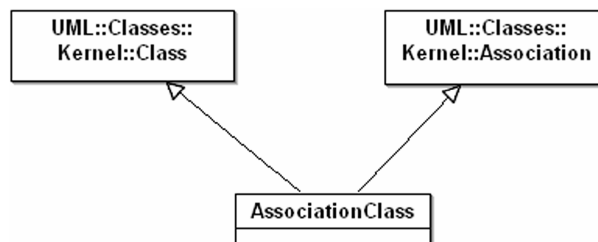

Figure 8 – Representation of the concept association class used to model classes (from the UML superstructure)

A dependency relationship, represented in Figure 9 by the *Dependency* class, indicates a supplier-client relationship between elements of the class model. A change in the supplier element may have impacts on the client element. This kind of relationship does not have impacts at runtime, it only has meaning in terms of the model and not its instances. The element on the tail of the association is the client element and depends on the supplier element situated on the tip of the arrow representing the association. A realization relationship, represented in Figure 9 by the *Realization* class, indicates that the elements on the tip of the arrow representing the relationship are realized by the elements on the tail of the arrow.

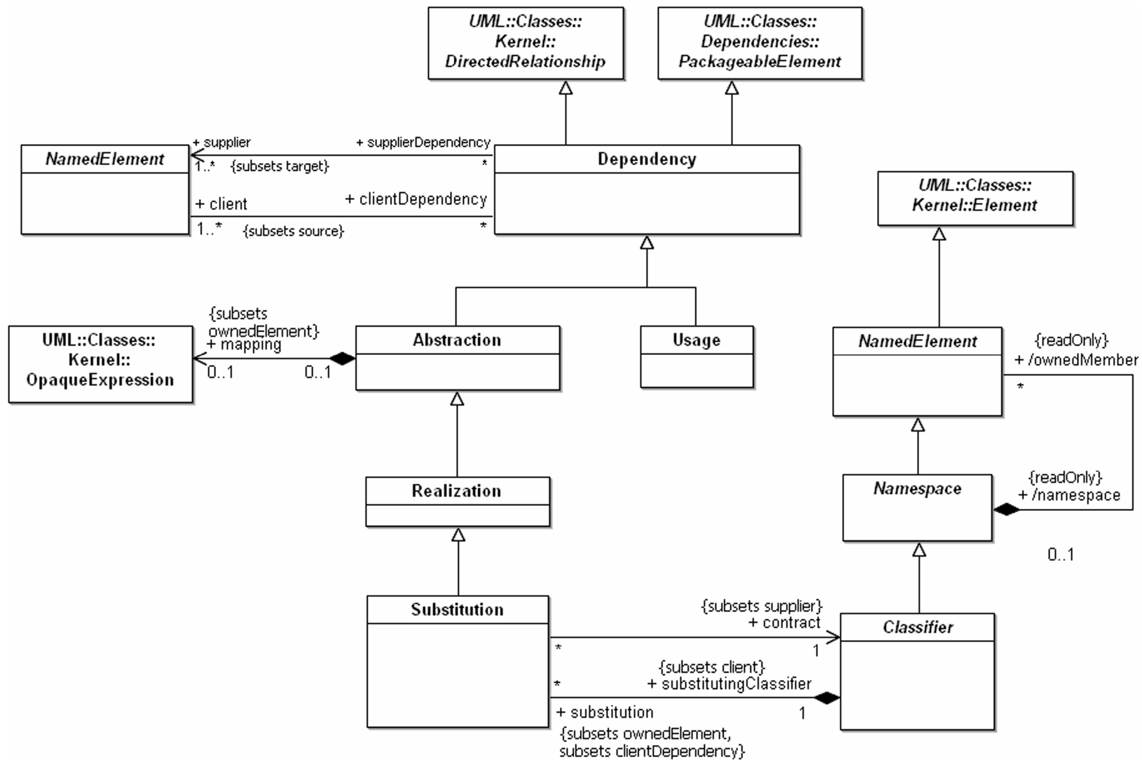

Figure 9 – Representation of the concepts dependency and realization used to model classes (from the UML superstructure)

A generalization relationship, represented in Figure 10 by the *Generalization* class, means that each instance of the specific element on the tail of the arrow that represents the relationship is also an instance of the general element on the tip of the arrow. So, the characteristics of the general element are also characteristics of the specific elements.

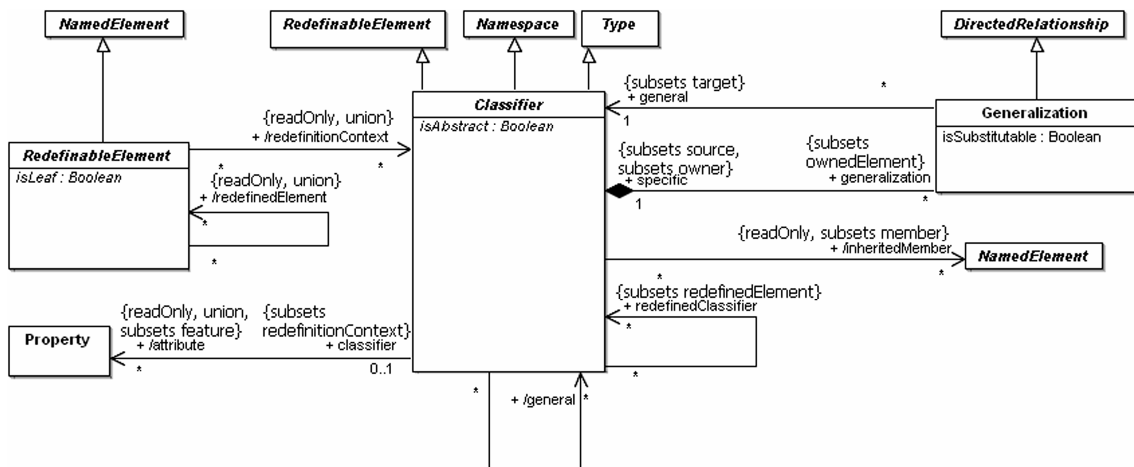

Figure 10 – Representation of the concept generalization used to model classes (from the UML superstructure)

An interface, represented in Figure 11 by the *Interface* class, refers to a set of public access characteristics and behaviours that are part of a service offered to the client of the specified system. Interfaces are implemented through classes, since they cannot be implemented directly.

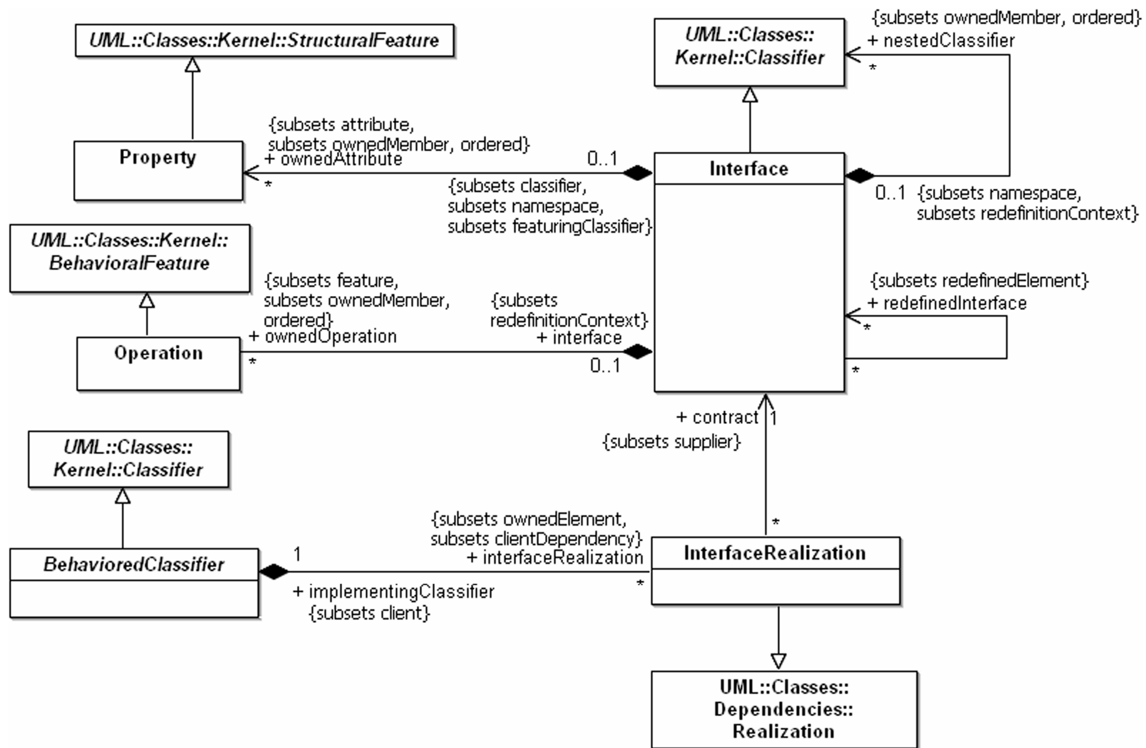

Figure 11 – Representation of the concept interface used to model classes (from the UML superstructure)

Activity diagrams emphasise the sequence of operations and conditions on the occurrence of those operations, both associated with each use case or behaviour of the system. An action, represented in Figure 12 by the *Action* class, represents an operation.

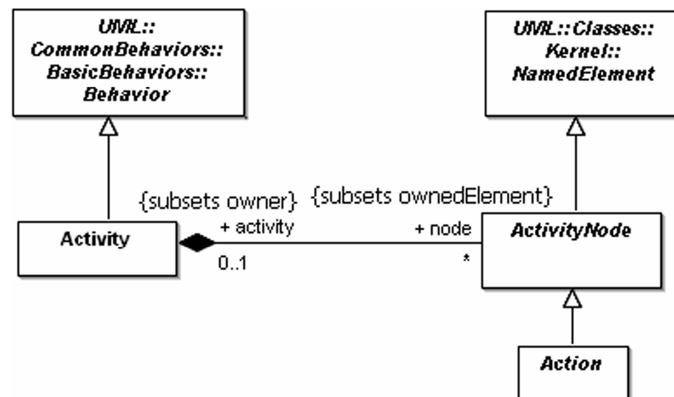

Figure 12 – Representation of the concept action used to model activities (from the UML superstructure)

An initial node is the starting point for the execution of an activity. If an activity has multiple initial nodes, then it means that the activity, when invoked, starts several flows. An activity final node is the element that ends the activity. Particularly, it ends all actions in execution at the time. Both initial node and activity final node are control nodes and are represented in Figure 13 by the *InitialNode* and *ActivityFinalNode* classes.

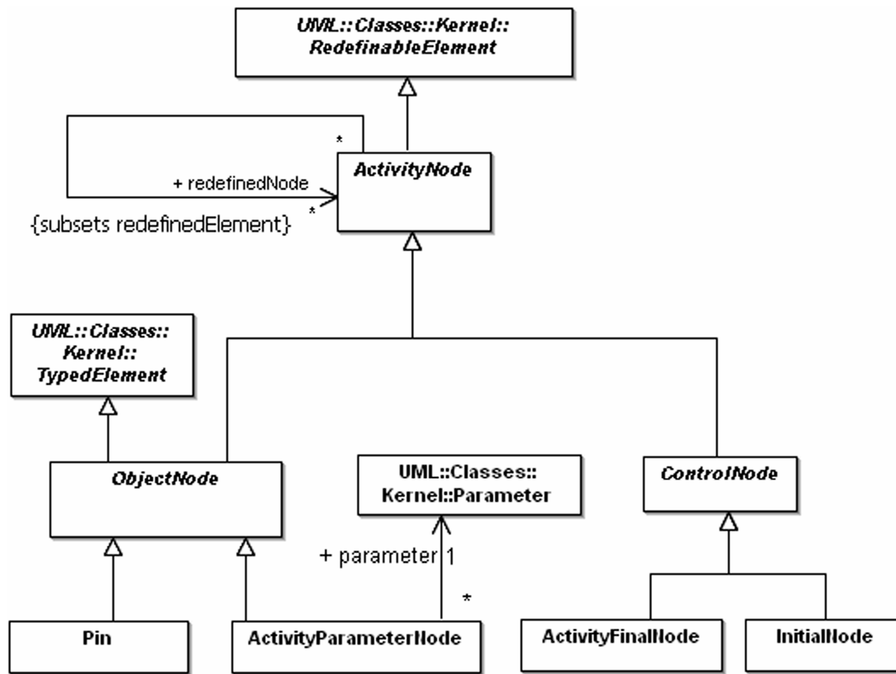

Figure 13 – Representation of the concepts activity final node and initial node used to model activities (from the UML superstructure)

Control flow, represented in Figure 14 by the *ControlFlow* class, is the element that, basically, connects two activities (its notation is an arrow).

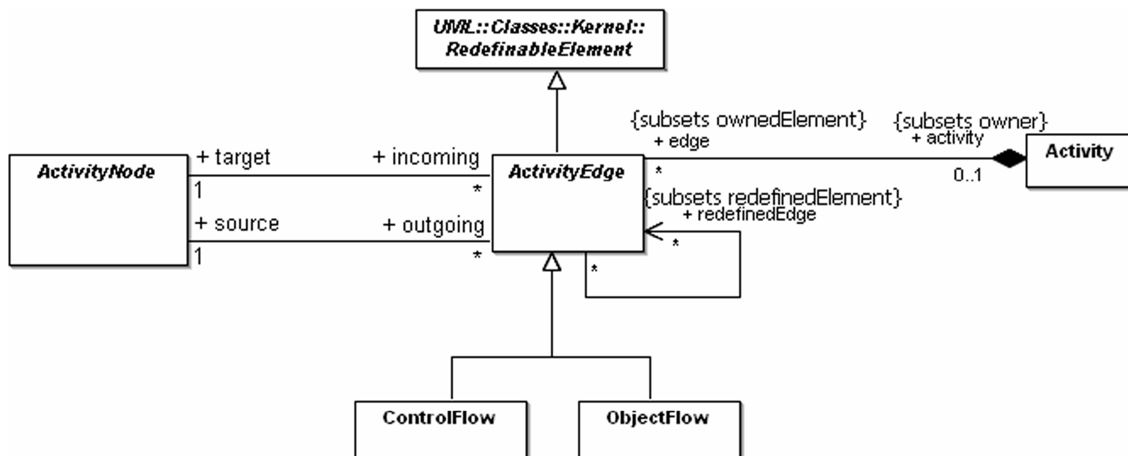

Figure 14 – Representation of the concept control flow used to model activities (from the UML superstructure)

An activity partition, represented in Figure 15 by the *ActivityPartition* class, is a compartment in the diagram that contains other elements of the activity diagram organized according to the name of the activity partition. That name may refer to an actor, for instance.

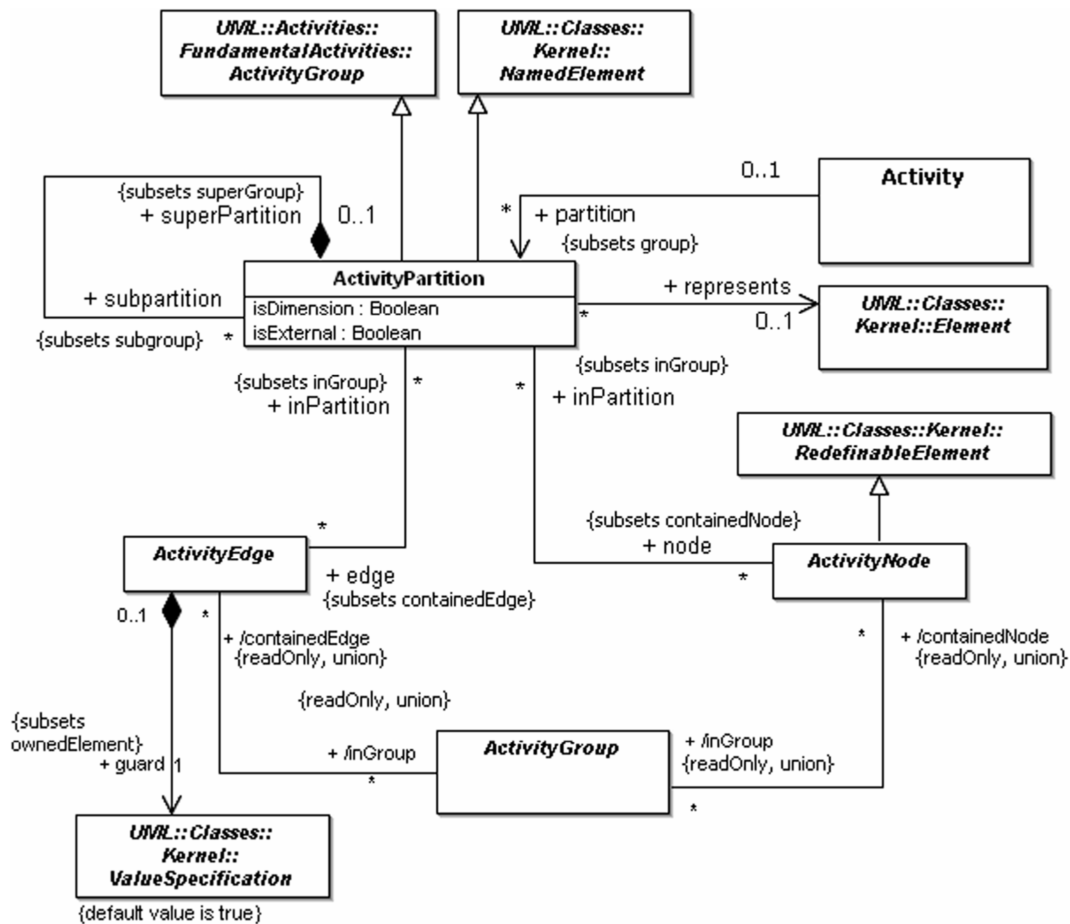

Figure 15 – Representation of the concept activity partition used to model activities (from the UML superstructure)

The concepts decision node, merge node, fork node and join node, used to model activities, are shown in Figure 16. A decision node (its notation is a lozenge) represents a decision. The output of an action entering a decision node will transit as input to a subsequent action according to the result of the decision, respecting the guard conditions the same decision presupposes. The software engineer shall have the concern of establishing decisions with mutually exclusive guard conditions, so that any output of any action fulfils only one of those conditions. The opposite of a decision node is a merge node (its notation is also a lozenge). It does not represent a synchronization of various flows but rather states that the same subsequent node can be entered by various alternative (not concurrent) flows. The difference between a decision node and a merge node is that the first one has only one incoming control flow and the last one has only one outgoing control flow. In the opposite extremes, decision node and merge node have more than one outgoing control flow and more than one incoming control flow, respectively. A fork node and a join node are related to synchronization. The fork node is the point in which a flow is split up into various concurrent flows. The join node is the point in which sets of actions belonging to concurrent execution

flows are synchronized. The notation of the fork and the join nodes is similar to the notation of decision and merge nodes except that in this case the shape is not a lozenge but rather a black bar.

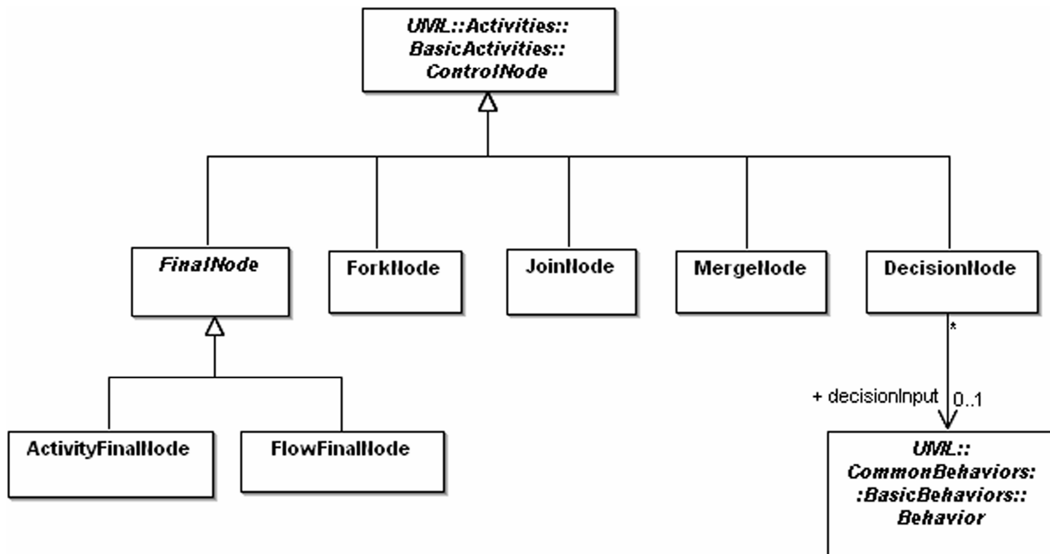

Figure 16 – Representation of the concepts fork node, join node, merge node and decision node used to model activities (from the UML superstructure)

At last, the state machine diagram allows modelling state transitions of objects belonging to a system. The concepts used to model state machines are shown in Figure 17. These state transitions are triggered by system's behaviours. A state represents a period of time in the object's lifetime. A state is activated when a transition enters the state and deactivated when the same or another transition leaves the state. An initial pseudostate, represented in Figure 17 by the *Pseudostate* class (in this case the attribute *kind* must have the value *initial*), is the element from which a unique transition to the default state is triggered. The default state is the first of the various states in the diagram. There can be only one initial pseudostate per diagram. The already mentioned unique transition to the default state has no associated guard condition. The opposite of an initial pseudostate is a final state, which is the last element of the diagram. A transition, in the context of a state diagram, is a relationship between two nodes in a state machine diagram. Each transition has a guard condition (that must appear between square brackets), which is evaluated before the transition's triggering. Guard conditions must only produce effects on the object's state at stake and no other effects besides those ones. A choice pseudostate, represented in Figure 17 by the *Pseudostate* class (in this case the attribute *kind* must have the value *choice*), is the element that allows evaluating the guard conditions of the different possible transitions entering it. If more than one of the evaluated guard conditions is true, one of those conditions is arbitrarily chosen. If

none of the conditions is satisfied, then the model was incorrectly conceived, which can be solved with an *else* guard condition.

Figure 17 – Representation of the concepts state, pseudostate, final state and transition used to model state machines (from the UML superstructure)

3.4. Using Modelling Environments to Configure Instances of UML Concepts with Domain-Specific Concepts

UML is a very powerful modelling language. But it is not enough to understand UML syntax and semantics, as it has just been done in this dissertation, if the goal is to conceive a DSL. This is just the first step in the process. It is important to consider its extension mechanisms in order to accomplish specific domain compliance. In that sense, stereotyping, as previously mentioned in this chapter, is the topic being analyzed in this section.

Fontoura, *et al.* [2000] created the UML-F profile. This profile is concerned with product line framework definition. UML-F is no more than a domain-specific profile of UML. Fontoura, *et al.* [2000] faced frameworks as sets of interoperating components which interact by means of cooperation patterns. Specialized forms of those components are implemented through application-specific code. If the metamodelling environment does not

support code generation of specialized code, UML-F allows defining the variation points/hot-spots/hooks from which specialized code, or application-specific code, can be written afterwards.

UML does not contain any domain-specific concepts, although it contains mechanisms which allow defining those concepts. Stereotypes allow customizing UML to become a language usable in a specific domain. As Fontoura*, et al.* [2000] and as France and Rumpe [2007] stated, stereotypes are specialized for a particular application domain. An example approach to stereotype class diagrams is now given as a way to discuss possible uses of stereotypes in the instantiation of a metamodel for the Primavera ERP sales domain.

UML originally provides the concept of class. But class in the context of the Primavera ERP sales domain can be applied to numerous domain concepts. The first step to define a DSL for this domain, using stereotypes, is to isolate the domain concepts that can be implemented as classes in the implementation language to be used. Since the Primavera ERP sales module is handled by sales people and the information handled can be about customers, sales person and customer can be two of the domain-specific concepts needed to be isolated for stereotypes' definition. Both of them are going to be implemented as classes; yet, class is very general compared with sales person or customer in the context of the Primavera ERP sales domain.

The difference between the definition of a class and the definition of domain-specific classes, like sales person or customer, is that class may have attributes and operations but, for example, sales person must have specific attributes, like an employee number, and specific operations, like *orderToSupplier()*. Specific attributes and operations of domain concepts must be defined as well as the domain concepts themselves.

The different variants of the Primavera ERP may require specialized forms of the sales person and/or customer classes. In this case, superclasses must be taken into consideration. That is the approach used to deal with commonality among variants of the Primavera ERP, by defining shared attributes and operations among variants of the software solution and then having the specialized ones only in the specialized forms of those classes. Besides different variants of the Primavera ERP (e.g. free, paid or demonstration), different software products of Primavera can also be considered for commonality definition.

The new metamodelling infrastructure of the Primavera ERP for the sales domain with UML notation must be defined in its syntactic part [Fontoura*, et al.*, 2000; Pree*, et al.*, 2002]. Two example templates for the definition of the domain concepts of sales person and of

customer mentioned previously (at the common syntax and at the specialized syntax levels) are presented in Table 5 and in Table 6.

Table 5 – Common syntax of two possible Primavera ERP concepts for the sales domain

Super domain concept	Super stereotype	Common attributes	Common operations
ERP super sales person	*«erpSuperSalesPerson»*	employee number	*orderToSupplier()*
		employee name	…
		…	…
ERP super customer	*«erpSuperCustomer»*	customer number	*orderToSalesPerson()*
		customer name	…
		…	…

Table 6 – Example of specialized syntax of two possible Primavera ERP concepts for the sales domain

Domain concept	Stereotype	Specialized attributes	Specialized operations
Paid ERP sales person	*«paidErpSalesPerson»*	photo	*insertPhoto()*
		-	-
Paid ERP customer	*«paidErpCustomer»*	-	*sendDemo()*
		-	-

Table 5 exemplifies the syntax definition of two superclasses. An example of two attributes and an operation is given for each (more could be added). The case could be of having a superclass with no attributes or with no operations. The meaning of a superclass is the same as the meaning of extensible classes defined by Fontoura*, et al.* [2000]: classes which are common to more than one application of the SPL (therefore, which are common inside a domain, the domain of the SPL) and that may require the addition of new operations for at least one of those applications. When modelling the classes of sales person and of customer, the corresponding stereotypes defined in the table must be used. The prefix *Super* is used in order to distinguish superclasses from subclasses.

Table 6 exemplifies the syntax of two specialized stereotypes, one for each of the two super domain concepts presented in Table 5. The combination of attributes and operations follows the same schema as the one mentioned for the superclasses, however, the meaning of those attributes and operations is not the same: the attributes and operations of subclasses define the application specificity towards the domain commonality defined by superclasses' attributes and operations.

The Primavera SPL could include all of Primavera's products (all specializations of the Primavera ERP, which are the Primavera Executive, the Primavera Professional, the Primavera Professional Starter, the Primavera Construction, the Primavera Industry, the Primavera AP, the Primavera AP POCAL and the Primavera Express, among other products) under the justification that all of them share the common features of business software. Another perspective is to consider that all variants of the Primavera ERP (the Primavera ERP paid variant, the Primavera ERP free variant, the Primavera ERP demonstration variant) form a SPL because they are a family of products sharing common features of business software. Whereas super domain concepts expressed in Table 5 affect more than one application of the ERP SPL (all ERP variants, for example), the domain concepts at the level of subclasses, expressed in Table 6, affect just one application of the ERP SPL (its paid variant, for instance). That is why the prefix *paidErp* is traversal to all stereotypes' names in Table 6.

Sales person and Customer are instances of Actor, a meta-domain concept, thus, a concept one level above the level of superclasses. This meta-domain concept can be applied to several domains.

Together, domain concepts, super domain concepts and meta-domain concepts define the metamodelling hierarchy to be used when metamodelling the sales domain of the Primavera ERP solution. The stereotyping metamodelling hierarchy is depicted in Figure 18. It distinguishes the concepts which reside at the metamodelling level (domain concepts and super domain concepts) from those which reside at the meta-metamodelling level or M3 layer (meta-domain concepts). Stereotypes are defined at the M2 level (the metamodelling level) and are used at the M1 level (the modelling level, where models are conceived). The concepts residing at the M3 level define concepts that can be used by the whole ERP SPL as well as by any other SPL. This is the only set of concepts not specific to a domain. The stereotypes at the super domain concepts are super stereotypes when compared with the stereotypes at the domain concepts level.

The possible values of attributes may be defined at this point. It is an approach to define at the metamodelling level the standardized values for attributes to be used when modelling the software solution. Therefore, the work of analysis is completely done when metamodelling: syntax, possible values and metamodelling hierarchy are pretty well defined.

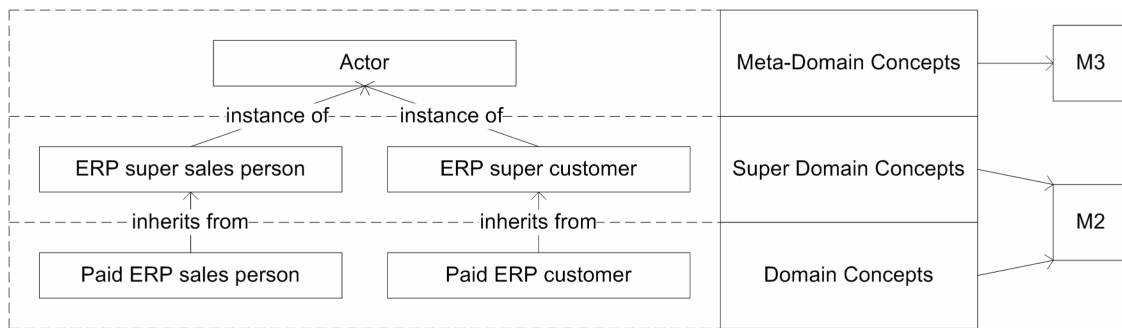

Figure 18 – Stereotyping metamodelling hierarchy

Generation tools can be used to create superclasses and subclasses in the code form and ask for the implementation of each one of the operations defined for both of them.

The stereotyping approach just presented has to be supported by the metamodelling environments. The domain engineer must be able to define the stereotypes and possible values for the attributes in the metamodelling environment. Then, the ideal situation for the software engineer would be to have an available set of stereotypes, attributes and respective possible values to use within the modelling environment. That way, the analysis work performed by the domain engineer would be easily reused by the software engineer. As far as the software developer is concerned, he should have the guidance from the code generation tool in order to know in which points to implement specific behaviour defined for the different application scenarios delimited by the domain engineer and designed by the software engineer.

Stereotypes' notation has been defined in this section. The next step would be to consider all the different variants of the Primavera ERP and define the domain concepts or the meta-domain concepts in each one of the three levels of the metamodelling hierarchy along with the respective stereotypes. During the design phase, using a modelling environment, the software engineer could apply those stereotypes created by the domain engineer to classes and those classes would acquire immediately the attributes and operations defined for the corresponding stereotypes.

3.5. Conclusions

This chapter of the dissertation developed specifically the metamodelling subject. The development cycle of metamodelling environments exposed the process of developing a particular software solution, which is a metamodelling environment. The also called meta-design environment was the software solution chosen owing to the fact that it is the primary environment within which the process of developing a DSL is performed in this dissertation.

Although the three roles are known, the responsibilities assumed by specially the software engineer and the software developer may not be the usual ones. The software engineer is often confused with the software developer and, so, both professionals' responsibilities may overlap and even be the same. The important aspect here is to think of roles or responsibilities instead of nomenclatures. The reason for these responsibilities being divided the way they are is discussed later in this dissertation.

Besides the equivalence between stereotyped class diagrams and DSLs, a view on the UML metamodel was presented and is going to be used in the DSLs' definition in chapter 4 of this dissertation.

The semantics of the different views or diagrams is now going to be resumed. The use case diagram models the functionalities of the system, in this case the ERP, accessible by actors through interfaces. Use cases are divided into various operations. The class diagram models the structure of the system in terms of objects handled by it, as an object-oriented model it is. The activity diagram models the operations of the system related to a particular use case or method. Each operation corresponds to an action performed by a particular actor in the context of the use case at stake. The state machine diagram models the states of an object during its lifecycle.

An approach used in the creation of the DSLs in the next chapter was described. The approach is about the definition of a SPL, the ERP SPL, and the definition of the stereotypes to be used when conceiving models with the DSLs in the modelling environment. The basis for the definition of the stereotypes was delineated in the stereotyping metamodelling hierarchy. The goal of the stereotypes is to configure instances of UML concepts with domain-specific concepts handled by the domain-specific application which is the Primavera Express.

4. From Metamodelling DSLs Inspired by UML to Designing Domain-Specific Models

This chapter is devoted to the demonstration case and its goal is to validate, with some mock-ups, the models conceived with a metamodelling approach for a part of the sales domain of Primavera ERP solution. At the end, a reflection is made upon the adopted metamodelling approach. The flow is from a first experimentation with Microsoft DSL Tools (the tool used to model at both levels M2 and M1) to the final metamodels and models for the sales domain of Primavera ERP solution.

4.1. Introduction

Microsoft DSL Tools [Bråthen, 2005; Microsoft Corporation, 2007] are deployed by Microsoft with Visual Studio SDK (Software Development Kit) 2005[9]. The tool allows creating DSVLs and generates code automatically (with the possibility of customization) from models conceived with the DSVLs.

---

[9] The version of Visual Studio SDK used was 4.0.

Microsoft DSL Tools allow the creation of graphical languages and the generation of code from the models conceived with those graphical languages. A graphical language is defined through a domain model and is conceived in an environment called *DSL Designer*, whereas models are conceived from those graphical languages in an environment which can be called *DSL Experimental Designer*. The domain model has two distinct compartments: (1) a compartment for the elements of the domain model and (2) a compartment for the graphical notation of the domain model elements. The graphical notation determines the graphical shape of an element instantiated in the *Experimental Designer*.

Files with extension *.dsl* define the syntax of the language. Each of these files contains a model. Code generators are files expressed in a text template language (a text template is a file with text blocks and control logic; whenever a transformation is run over the text template, the control logic combines the text blocks with the data in the model to produce the generated code). When the *Transform All Templates* action is performed (see Figure 19), all code generators are run and the code is generated either from the *Designer* (the environment where the domain model is conceived), depicted in Figure 20, or from the *Experimental Designer* (the environment where models are conceived), depicted in Figure 21.

Text templates, which are files written in a text template language, can iterate over the model created in the *Experimental Designer* (which is basically an instance of the domain model) to generate code in the chosen GPL. That flow can be seen in Figure 22, the layered modelling architecture of Microsoft DSL Tools.

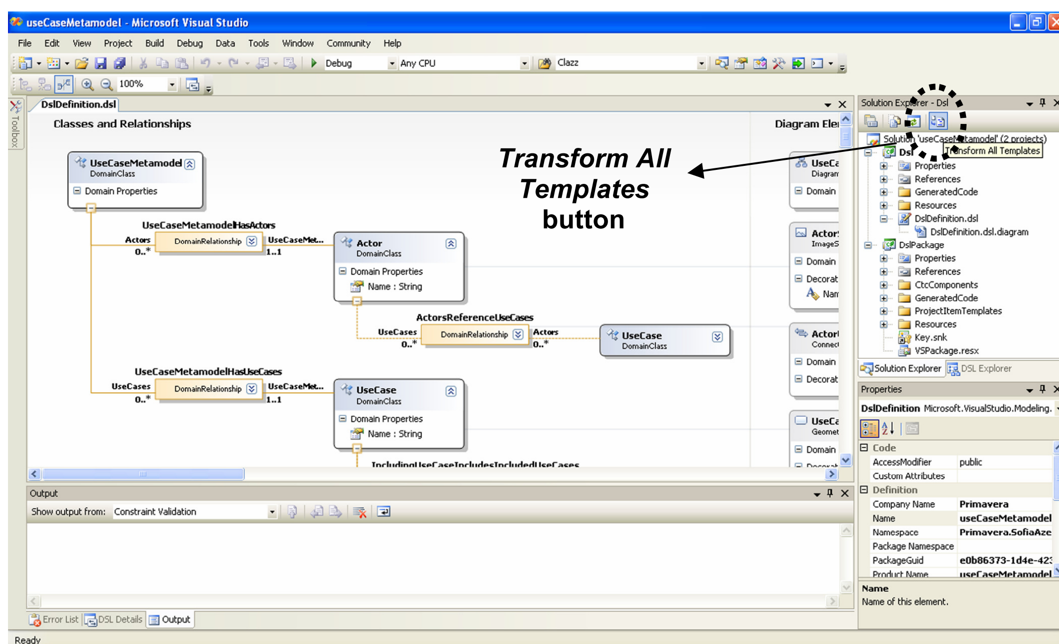

Figure 19 – Transform All Templates button

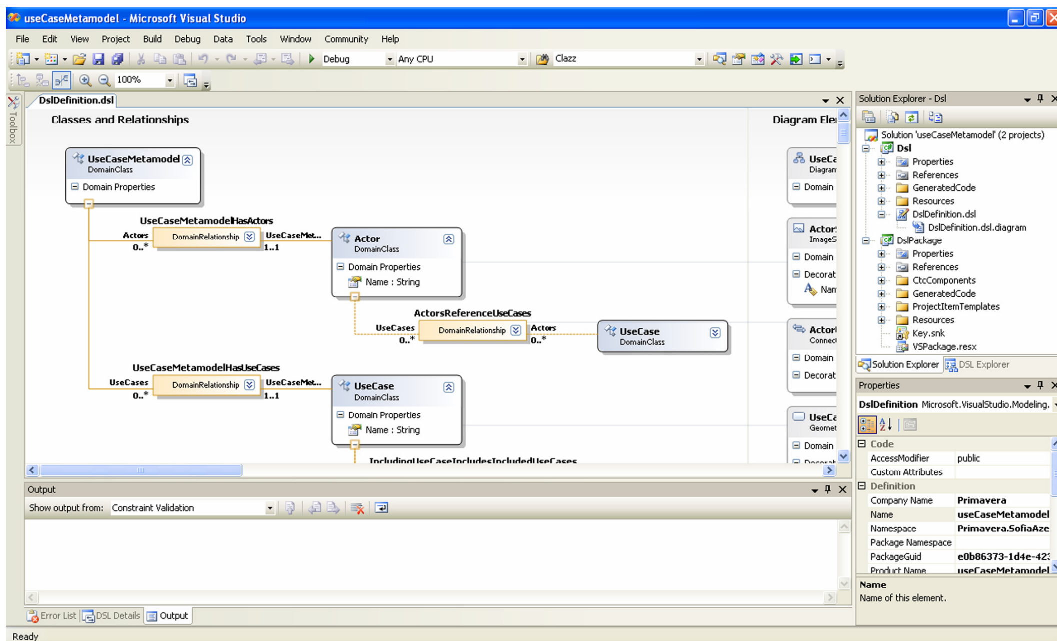

Figure 20 – The DSL Designer

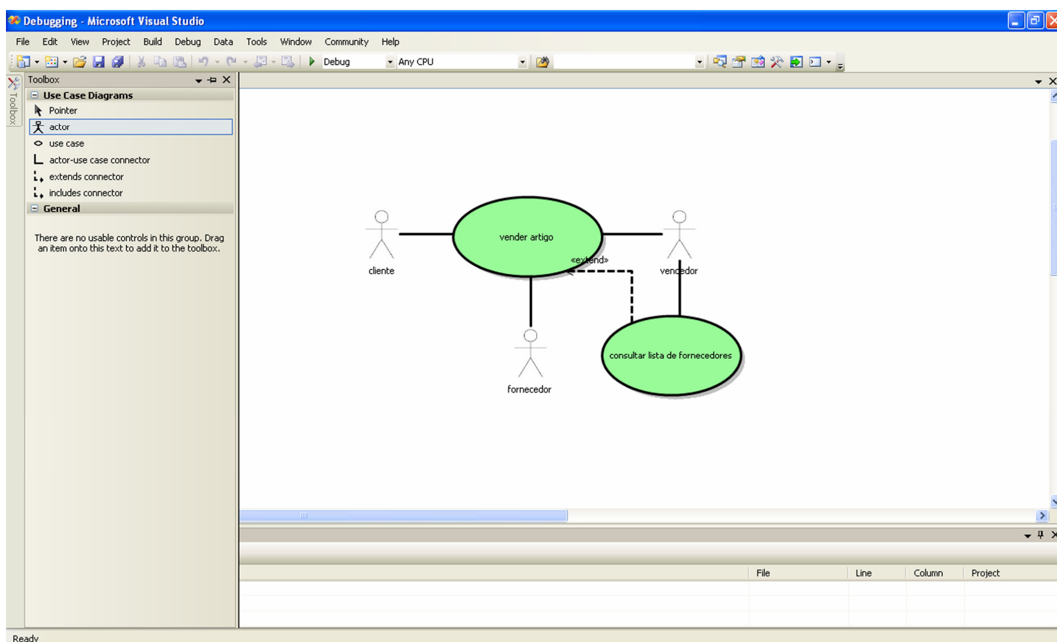

Figure 21 – The DSL Experimental Designer

This architecture is divided into three environments (metamodelling environment, modelling environment and implementation) to which correspond, respectively, the levels M2, M1 and M0 of the Four-Layer Architecture of UML. The metamodels, or domain models, reside at the metamodelling environment; the models, or diagrams, reside at the modelling environment; and the generated code resides at the implementation level.

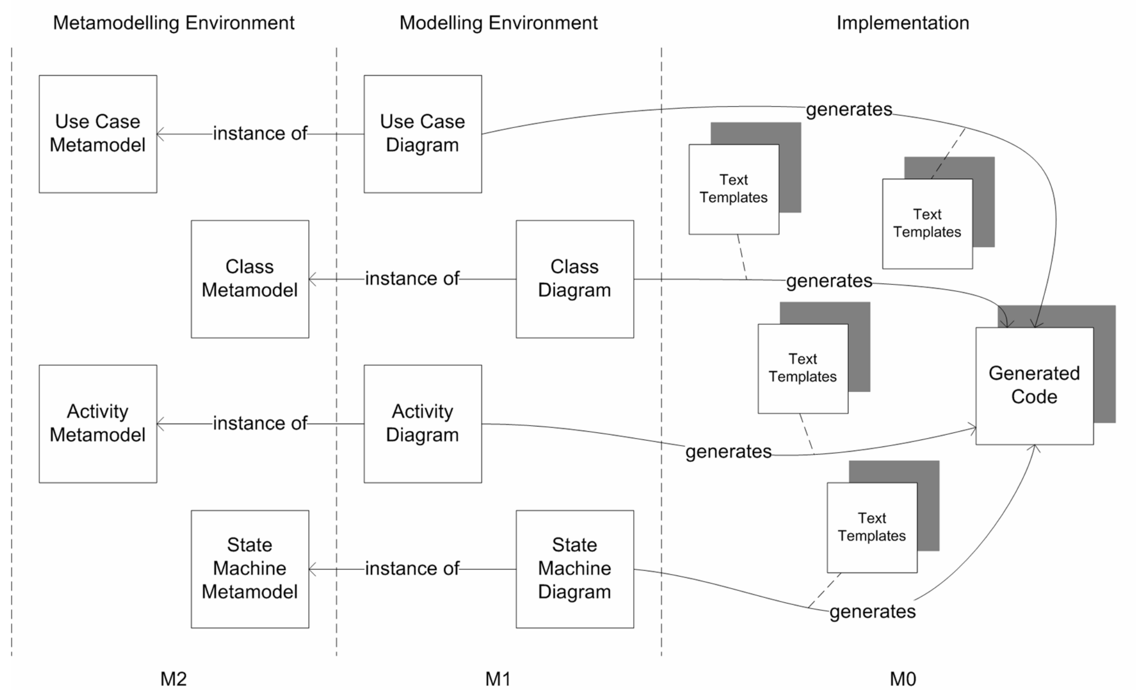

Figure 22 – Layered modelling architecture of Microsoft DSL Tools

The solution to a DSVL for a part of the Primavera ERP (which can be called the sales module) using Microsoft DSL Tools is composed by a graphical language allowing the instantiation of models for the sales module of the Primavera ERP. Using the graphical language is, in fact, using a metamodel for the sales domain. The usefulness of this metamodel resides in the fact that a syntactic basis is of strong reutilization power when it comes to instantiating models in the context of sales (as a subsystem of the Primavera ERP) in a multiview perspective [Machado*, et al.*, 2005b].

4.2. First Experimentation with Microsoft DSL Tools

This section is dedicated to the first experimentation performed with Microsoft DSL Tools. The tool's infrastructure and functionality are explained throughout the section and followed by examples related to the context defined in Chapter 1. The result of this experimentation is shown at the end of the section and is a simple example of how DSL Tools can be used to conceive DSLs for the sales domain of the Primavera ERP software solution.

4.2.1. The Example of Two Graphical Languages: Class Diagram and State Machine Diagram

In order to create a DSL designer with Microsoft DSL Tools it is necessary to create a new project in Visual Studio for a DSL designer. This way it is created a Visual Studio's

solution for an environment where DSLs can be designed. The solution includes a domain model and a visual designer for the new language.

Figure 23 depicts the definition of a metamodel for the general business objects of the Primavera ERP [Pereira, 2007]. On the left side (the domain model), we can see the types of specific objects of the Primavera ERP domain (element types; in this case, only the type *BusinessObject* is presented) and the relationships between them (relationship types, the orange elements). Connections with different multiplicities and properties exist between element types and relationship types, similar to a UML class diagram. The element type *Attribute* is going to be embedded in the element type *BusinessObject*, that's why a relation is defined between both. Both *BusinessObject* and *Attribute* element types are under the element *GeneralBusinessObjects*, which means that those two element types will be subelements (equivalent to subclasses in the OO paradigm) of a superelement (equivalent to a superclass in the OO paradigm) which is the element *GeneralBusinessObjects*. This superelement is an abstraction that represents all the metamodel elements defined in the domain model. On the right side, we can see the notation's definition which will allow the graphical materialization of objects belonging to a specific element type or relationship type in a model to be built later on in the *Experimental Designer*, which will partially result from the compilation of the domain model and the notation's definition. Between the elements that determine the notation and both the element types and relationship types are defined mappings (straight line segments connecting both), which establish the connection between notation and element or relationship types in the model to be built. Still on the right side, there is the unique element type *Diagram* that represents an abstraction of the model (or diagram) as container of all notation's definition. This element type is an element type from the metamodel of *Designers*. The metamodel's defined notation and both element types and relationship types are represented in the Toolbox of the *Experimental Designer*.

Figure 24 shows another domain model. This one is concerned with elements of a possible state machine diagram for the Primavera ERP general business objects [Pereira, 2007]. The novelty here is the presence of a superelement type *StateMachineElement*, with which the element types *State*, *Choice*, *Initial* and *Final* have an inheritance relation. Another novelty is the presence of a relationship type, *StateMachineElementReferencesTarget*, which is established between elements of the same (element) type.

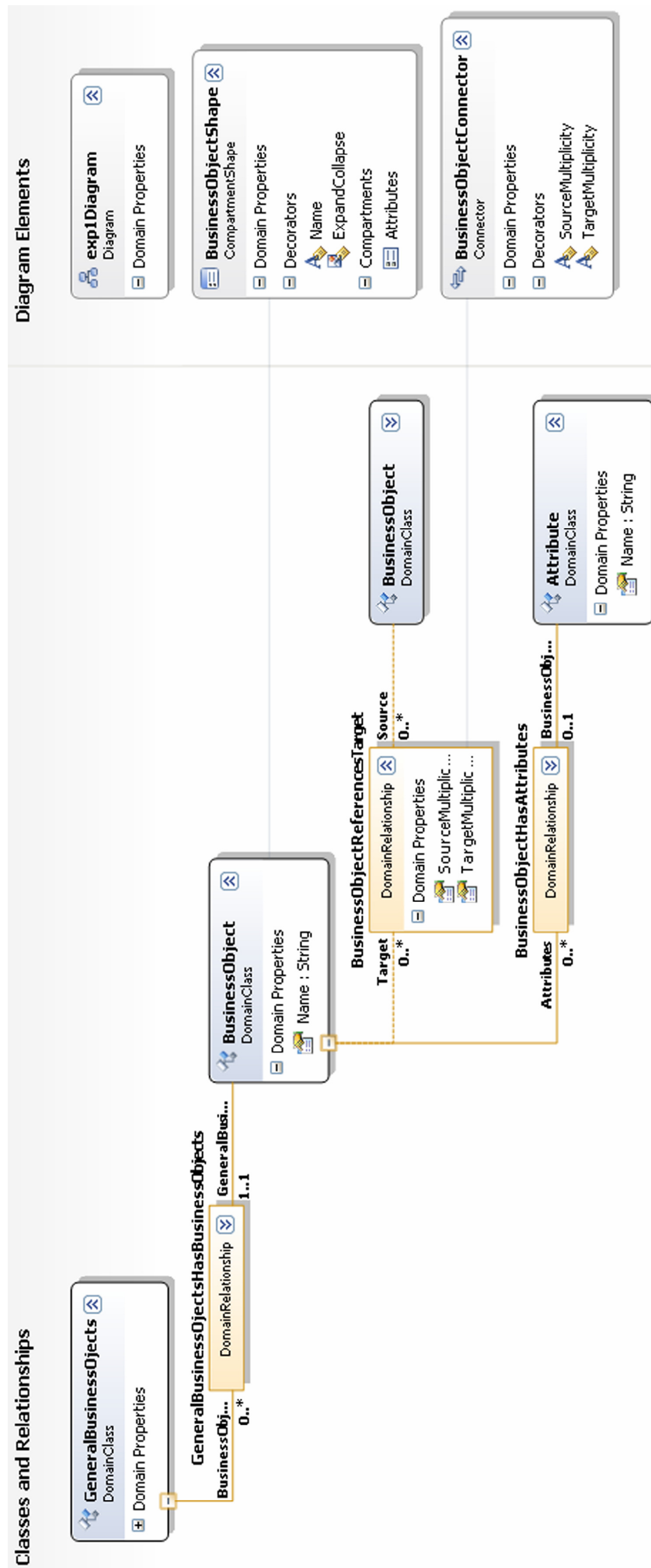

Figure 23 – The Primavera ERP general business objects domain model

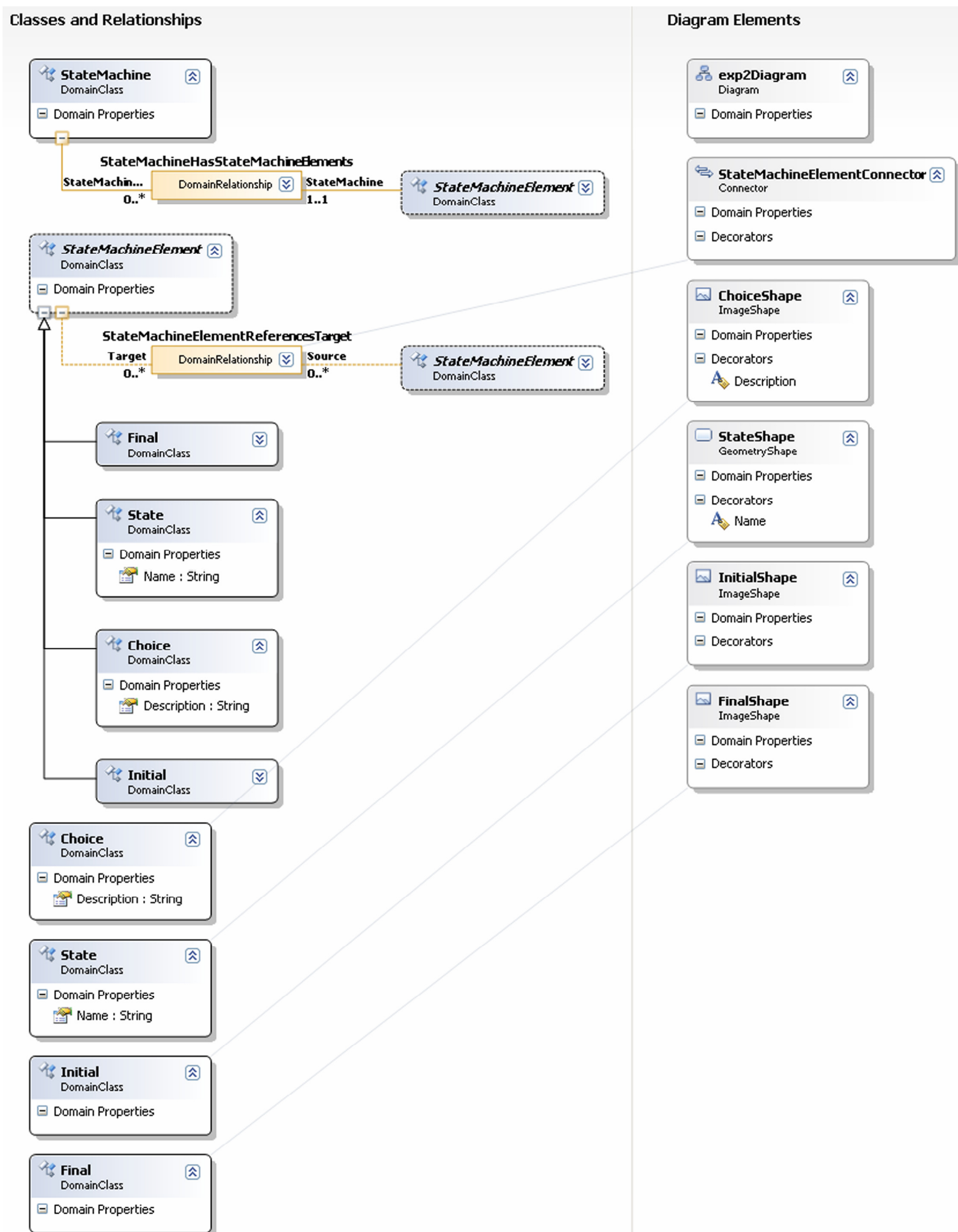

Figure 24 – Domain model for a possible state machine diagram regarding the Primavera ERP general business objects

Figure 25 illustrates an example of a model instantiated from the domain model depicted in Figure 23. The model is equivalent to a UML class diagram. If the name of the element on the left side, *a*, is replaced by *customer* and the name on the right side, *b*, is replaced by *sales person*, for instance, two classes of the sales module of the Primavera ERP are classified. *Attribute a* and *Attribute b* in *BusinessObject a* can be, for example, *id* and *name*. *SourceMultiplicity* and *TargetMultiplicity* may get values like 0..*, 0..1, among others. The

environment (or *Experimental Designer*) where the model in Figure 25 was conceived was partially provided by the compilation of code generated out of the respective domain model (Figure 23). The same applies to the rest of the pairs of models presented throughout the remainder of this document. Hence, in DSL Tools, domain models are conceived in one environment called the *Designer*, whereas models are conceived in another environment called the *Experimental Designer*, and for each pair of domain model and model in this dissertation there is a pair of *Designer* and *Experimental Designer*.

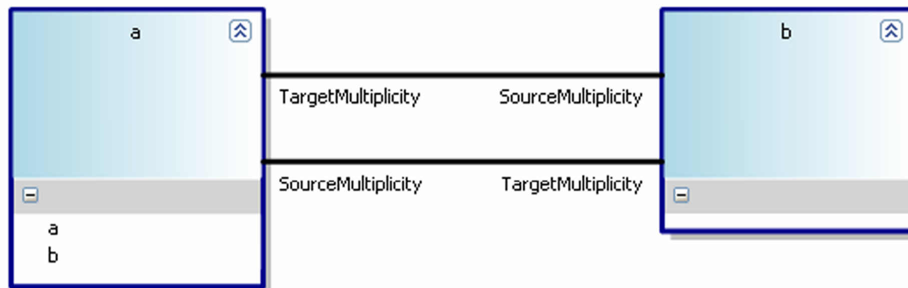

Figure 25 – Example of a model equivalent to a class diagram

Figure 26 depicts an example of a model instantiated from the domain model presented in Figure 24.

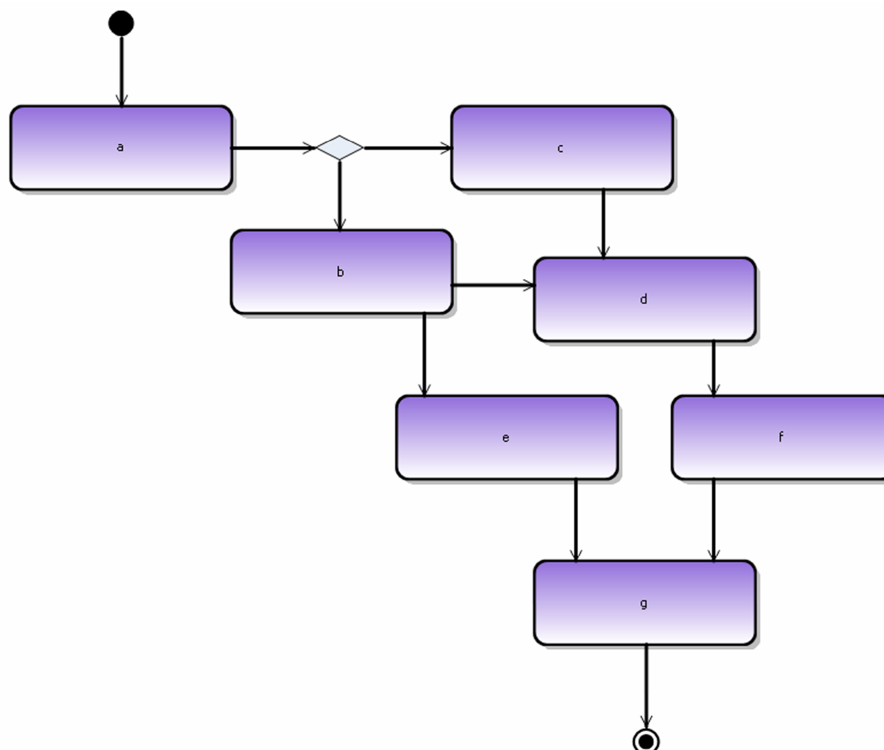

Figure 26 – Example of a model equivalent to a state machine diagram

The notation in this model is similar to the notation of a UML state machine diagram. It has a starting point and an ending point, object states and decision points (in this case, only one decision point). An example of a state machine diagram regarding the Primavera ERP

object called order would have, for instance, two states: *ordered* and *invoiced*. The way from *ordered* state to *invoiced* state would be triggered by the decision "Order handled by supplier and received at the store?"

4.2.2. The Generated Code

The automatic generation of code produced from the action of *Transform All Templates*, applied to the project containing the definition of the DSL, is described here. When that action is performed, the text templates (files with extention *.tt*) are going to automatically originate files (in this case) with extension *.cs*. The *.cs* files are generated from the metamodels, rather than from the models, as it was already mentioned in this dissertation (see Figure 27).

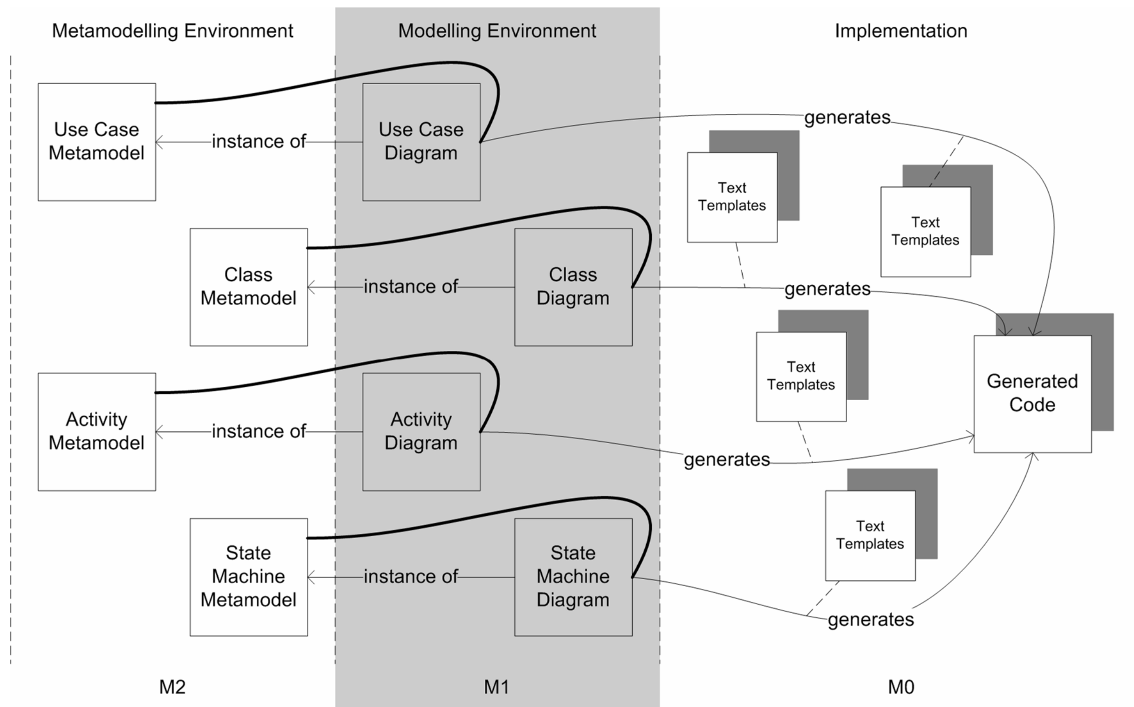

Figure 27 – Layered modelling architecture of Microsoft DSL Tools for the first experimentation with the tool

Figure 28 illustrates part of the file containing the domain classes regarding the domain model of the Primavera ERP general business objects. Figure 28 shows the signature of three classes: *GeneralBusinessObjects*, *BusinessObject* and *Attribute*. All of them are derived classes from the base class *ModelElement* (Visual Studio's class).

```
        public partial class GeneralBusinessOjects : DslModeling::ModelElement
        {
        }

        public partial class BusinessObject : DslModeling::ModelElement
        {
        }

        public partial class Attribute : DslModeling::ModelElement
        {
        }
```

Figure 28 – Excerpt of the file DomainClasses.cs containing generated code for the domain classes in the domain model of the Primavera ERP general business objects

Figure 29 shows part of the file containing the domain relationships regarding the domain model of the Primavera ERP general business objects. Figure 29 depicts the signature of three classes: *GeneralBusinessObjectsHasBusinessObjects*, *BusinessObjectReferencesTarget* and *BusinessObjectHasAttributes*. All of them are derived classes from the base class *ElementLink* (Visual Studio's class).

```
        public partial class GeneralBusinessOjectsHasBusinessObjects :
DslModeling::ElementLink
        {
        }

        public partial class BusinessObjectReferencesTarget : DslModeling::ElementLink
        {
        }

        public partial class BusinessObjectHasAttributes : DslModeling::ElementLink
        {
        }
```

Figure 29 – Excerpt of the file DomainRelationships.cs containing generated code for the domain relationships in the domain model of the Primavera ERP general business objects

Figure 30 shows part of the code that allows performing the get and set of the domain property *Name* from the class *BusinessObject*. The set of the *BusinessObject*'s *Name* is done using the method *SetValue*.

```
        private global::System.String namePropertyStorage = string.Empty;

        public global::System.String Name
        {
                get
                {
                        return namePropertyStorage;
                }
                set
                {
                        NamePropertyHandler.Instance.SetValue(this, value);
                }
        }
```

Figure 30 – Code that defines the domain property Name of the class BusinessObject

Figure 31 illustrates the get and set methods for the domain property *BusinessObjects* regarding the relationship *GeneralBusinessObjectsHasBusinessObjects*.

```
        public virtual GeneralBusinessOjects BusinessObjects
        {
                get
                {
                        return
(GeneralBusinessOjects)DslModeling::DomainRoleInfo.GetRolePlayer(this,
BusinessObjectsDomainRoleId);
                }
                set
                {
                        DslModeling::DomainRoleInfo.SetRolePlayer(this,
BusinessObjectsDomainRoleId, value);
                }
        }
```

Figure 31 – Code that defines the domain property BusinessObjects

Figure 32 depicts part of the file containing the domain classes regarding the domain model of a possible state machine diagram for the Primavera ERP general business objects. The figure shows the declaration of seven classes and these are: *StateMachine*, *StateMachineElement*, *Choice*, *State*, *Initial* and *Final*. The first of these two classes are derived from the base class *ModelElement* (Visual Studio's class), whereas the other four are derived from the base class *StateMachineElement*. The reason for this to happen is due to the fact that the superclass of the four classes, the class *StateMachineElement*, is in the definition of the DSL instead of being a Visual Studio's class.

```
        public partial class StateMachine : DslModeling::ModelElement
        {
        }

        public abstract partial class StateMachineElement : DslModeling::ModelElement
        {
        }

        public partial class Choice : StateMachineElement
        {
        }

        public partial class State : StateMachineElement
        {
        }

        public partial class Initial : StateMachineElement
        {
        }

        public partial class Final : StateMachineElement
        {
        }
```

Figure 32 – Excerpt of the file DomainClasses.cs containing generated code for the domain classes in the domain model of a possible state machine diagram regarding the Primavera ERP general business objects'

Figure 33 depicts part of the file containing the domain relationships present in the domain model of a possible state machine diagram for the Primavera ERP general business objects. Figure 33 shows the signature of two classes: *StateMachineHasStateMachineElements* and *StateMachineElementReferencesTarget*. Both are derived classes from the base class *ElementLink* (Visual Studio's class).

```
public partial class StateMachineHasStateMachineElements : DslModeling::ElementLink
{
}

public partial class StateMachineElementReferencesTarget : DslModeling::ElementLink
{
}
```

Figure 33 – Excerpt of the file DomainRelationships.cs containing generated code for the domain relationships in the domain model of a possible state machine diagram regarding the Primavera ERP general business objects'

4.3. Metamodelling with Microsoft DSL Tools Considering the UML Superstructure

This section of chapter 4 is about the first step to be taken when metamodelling DSLs, which is by considering the UML superstructure. Stereotypes are also used. Primarily, UML metamodel's concepts must be mapped into DSL Tools' concepts. Finally, the DSL has to be defined by means of a metamodel in a metamodelling environment and used to build UML models in a design environment (using stereotypes or the already defined domain concepts).

4.3.1. Mapping Some of DSL Tools' Concepts into UML Metamodel's Concepts

According to the specification of UML v2.1.1 superstructure [OMG, 2007b], the different elements of the UML metamodel presented in the superstructure and referred in the previous chapter of this dissertation are now mapped into the elements of the metamodel of Microsoft DSL Tools. These elements are depicted in Figure 34, the Toolbox of the metamodelling environment of Microsoft DSL Tools.

The types of elements from the UML metamodel, in this case only those related to the use case diagram, are represented in the Toolbox of the modelling environment shown in Figure 35.

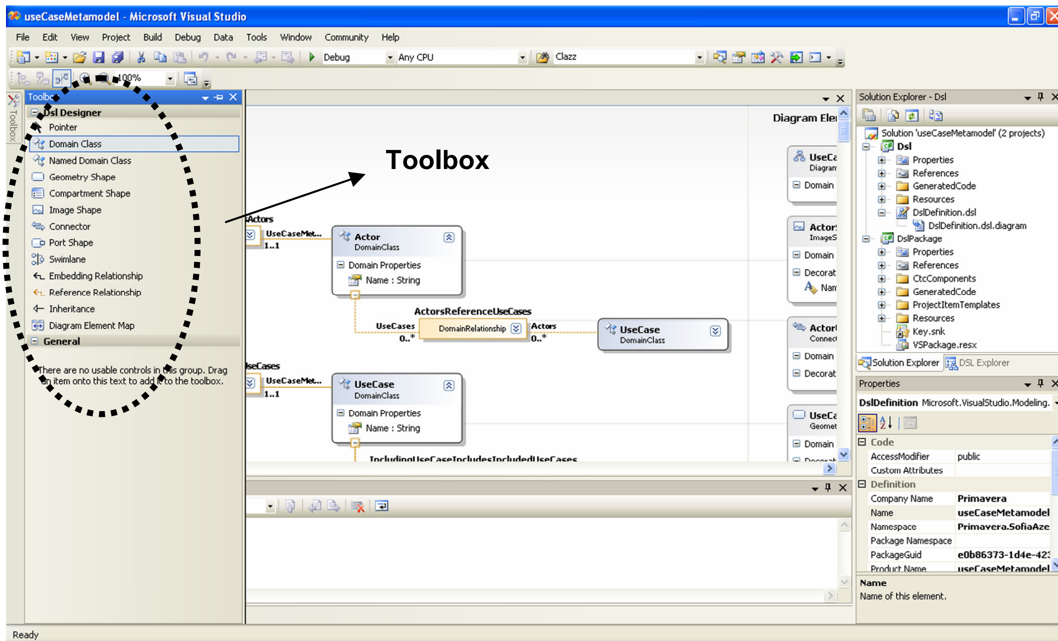

Figure 34 – Metamodelling environment's Toolbox

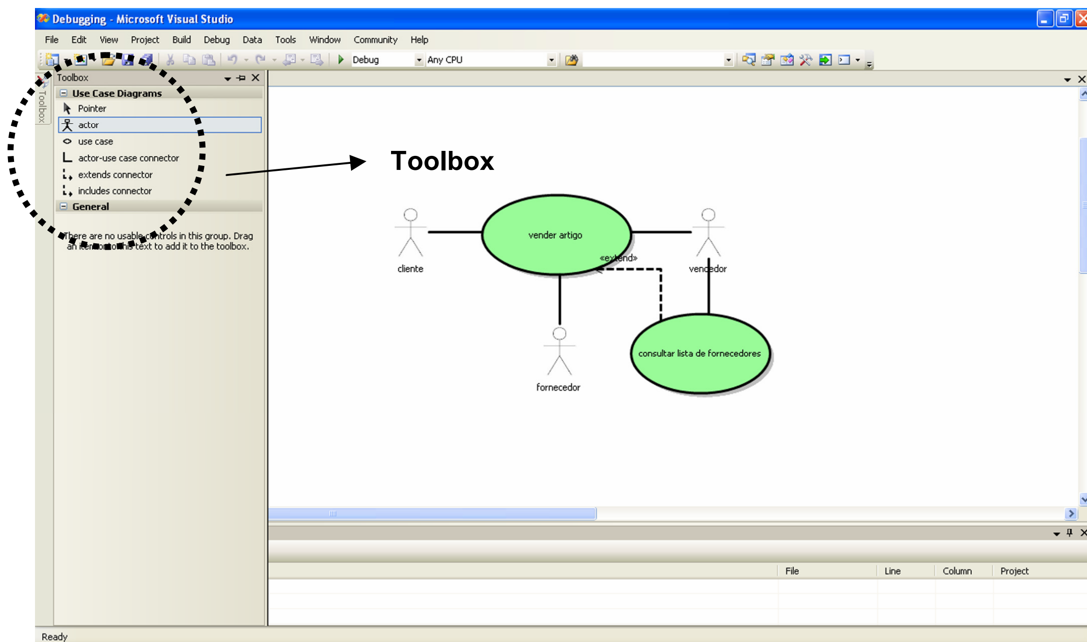

Figure 35 – Modelling environment's Toolbox

Table 7 shows the mapping between some types of elements from the UML metamodel and the types of elements from the metamodel of DSL Tools, which defines the syntax of the *Designers*. This kind of mapping was mentioned by Demir [2006] and by Sprinkle and Karsai [2004] as being necessary in a metamodelling approach.

Table 7 – Mapping between some types of elements from the UML metamodel and the types of elements from the metamodel of DSL Tools

Diagrams	Types of elements from the UML metamodel	Types of elements from the metamodel of DSL Tools
Use case diagram	Actor	ImageShape
	Use case	GeometryShape
	Include	Connector
	Extend	Connector
Class diagram	Class	CompartmentShape
	Attribute	N/A
	Operation	N/A
	Realization	Connector
	Composition	Connector
	Aggregation	Connector
	Unidirectional Association	Connector
	Bidirectional Association	Connector
	Generalization	Connector
	Dependency	Connector
Activity diagram	Activity Partition	Swimlane
	Decision/Merge Nodes	ImageShape
	Fork/Join Nodes	GeometryShape
	Action	GeometryShape
	Initial Node	ImageShape
	Final Node	ImageShape
	Control Flow	Connector
State machine diagram	Transition	Connector
	Choice Pseudostate	ImageShape
	State	GeometryShape
	Initial Pseudostate	ImageShape
	Final State	ImageShape

4.3.2. Metamodel for the Sales Domain of the Primavera ERP Solution

This subsection of the dissertation presents the metamodels that compose the Primavera ERP metamodel for the sales module and corresponding model examples. All domain models obey the mapping in Table 7. The metamodels were all conceived within DSMEs and represent the definition of the DSLs used to conceive each one of the models within each one of the DSDEs.

Figure 36 depicts the metamodel (or domain model) of the use case diagrams (like the one in Figure 37) conceived within the scope of the Primavera ERP metamodel for the sales domain. The domain class *UseCaseMetamodel* is the root of all domain model elements (domain classes, domain relationships and domain properties).

The left part of Figure 36 shows the metaclasses from which classes can be instantiated in the *Experimental Designer* when conceiving the use case diagram. Those metaclasses are: (1) the domain classes *Actor* and *UseCase*; (2) the domain relationships that can be instantiated in the diagram e.g. *ActorsReferenceUseCases*; (3) the domain relationships that cannot be instantiated in the diagram e.g. *UseCaseMetamodelHasActors*; (4) the domain properties *Name* of e.g. *Actor* and *IncludingUseCaseIncludesIncludedUseCases*. The domain relationships that cannot be instantiated in the diagram are used to express the containment of the domain classes that can be instantiated in the diagram by that diagram and, so, those invisible domain relationships don't have a corresponding diagram element on the right side of Figure 36.

The diagram elements determine the shapes of each of the domain classes and visible domain relationships in the model (with the respective decorators; these decorators allow to decorate the shape with the domain properties of the domain classes or the domain relationships to which the shape is associated with): in Figure 36 *ActorShape* determines the shape of *Actor*, *ActorUseCaseConnector* determines the shape of *ActorsReferenceUseCases*, and so on. *UseCaseDiagram* represents the diagram that can be built in the *Experimental Designer* as an instance of the domain model.

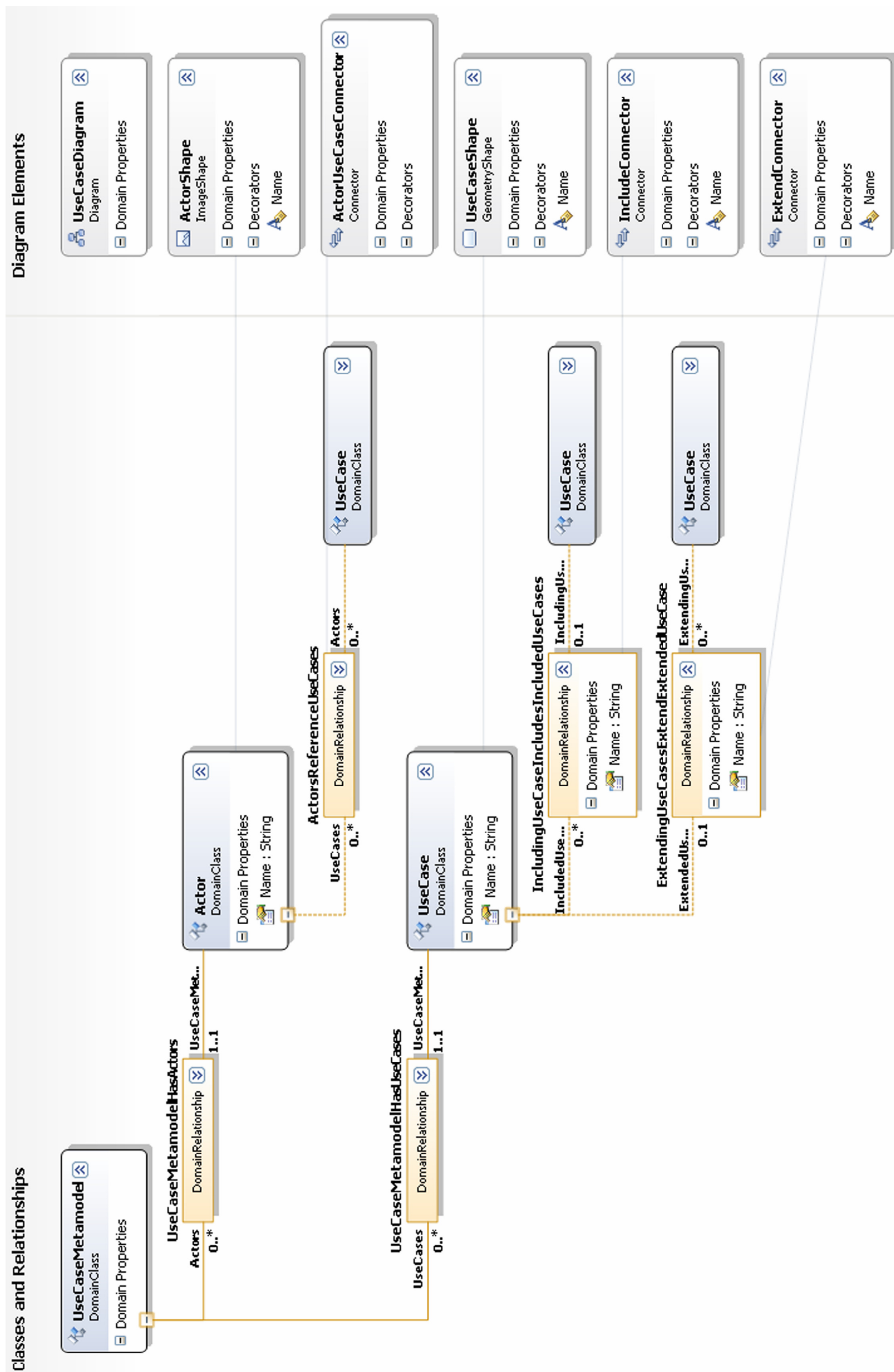

Figure 36 – Use case metamodel that is part of the Primavera ERP metamodel for the sales domain

Figure 37 shows a use case diagram with only two use cases (*vender artigo* (sell product) and *consultar lista de fornecedores* (consult suppliers list)) having an extend relationship between them. The use case diagram depicts two behaviours of the ERP accessible by the three actors through interfaces.

Three actors are in the use case diagram: *cliente* (customer), *vendedor* (sales person) and *fornecedor* (supplier). These actor names, as well as *artigo* (product) and *fornecedores* (suppliers), correspond to domain concepts like those mentioned in section 3.4. They allow using the metaclass *Actor* adapted to a domain-specific context. Stereotypes could have been used.

Regarding Figure 37 and establishing a comparison between model and metamodel, *cliente*, *vendedor* and *fornecedor* are all instances of the domain class *Actor*; *vender artigo* and *consultar lista de fornecedores* are both instances of the domain class *UseCase*. All their names are instances of the domain property *Name* of the respective domain class. The extend relationship is an instance of the domain relationship *ExtendingUseCasesExtendExtendedUseCase*. The *String «extend»* is an instance of the domain property *Name* of the domain relationship *ExtendingUseCasesExtendExtendedUseCase*.

The fact that *vendedor* in Figure 37 is associated with the two use cases is possible because in the metamodel there is the multiplicity of 0..* on the left side of the domain relationship *ActorsReferenceUseCases*, associated with the domain property *UseCases*, indicating that an actor may be associated with zero or more use cases. The same situation happens with use cases associated with actors.

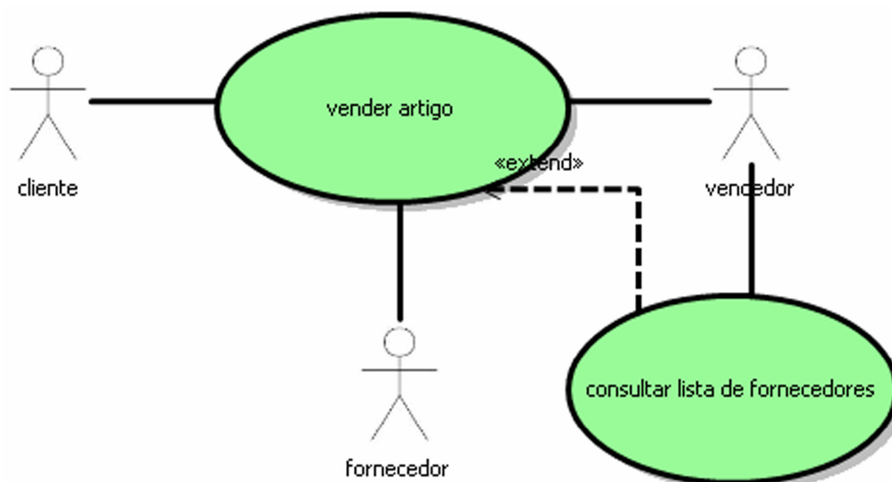

Figure 37 – Example of a use case diagram in the context of sales

Figure 38 illustrates the metamodel (or domain model) of the class diagrams (like the one in Figure 39) conceived within the scope of the Primavera ERP metamodel for the sales domain.

In terms of construction, the only difference between the metamodel in Figure 38 and the metamodel in Figure 36 is the existence of a super domain relationship, a super connector and a compartment shape. The super domain relationship is the abstract domain relationship called *SourceClassesReferenceTargetClasses*. The super connector is the super diagram element called *AssociationConnector*. The compartment shape is called *ClassShape*. The super domain relationship allows determining the domain properties of all domain relationships which have an inheritance relationship with the abstract domain relationship (e.g. the domain relationship *ComposedClassIsACompositionOfComposingClasses*). The super connector allows determining the decorators, one for each of those domain properties, associated with the diagram elements corresponding to each one of the domain relationships. All the four domain relationships mentioned above and all the corresponding diagram elements have an inheritance relationship with super domain relationship and with the super connector relationship, respectively, so, they inherit the properties and decorators of the super domain relationship and of the super connector, respectively. The compartment shape is a shape with compartments, as the name indicates. Those compartments will be shown in the shapes of *Class*' instances (*Class* is the domain class associated with the compartment shape). One of the compartments will contain *Attributes* (instances of the domain class *Attribute*) and the other one will contain *Operations* (instances of the domain class *Operation*). In order for this to be possible, the domain class *Class* is related to the domain classes *Attribute* and *Operation* through two distinct domain relationships (*ClassHasAttributes* and *ClassHasOperations*, respectively). The domain class *Class* is domain-specific if combined with the stereotypes mentioned in section 3.4.

Figure 39 depicts a class diagram in the context of the Primavera ERP sales domain. The diagram shows the business objects handled by the ERP.

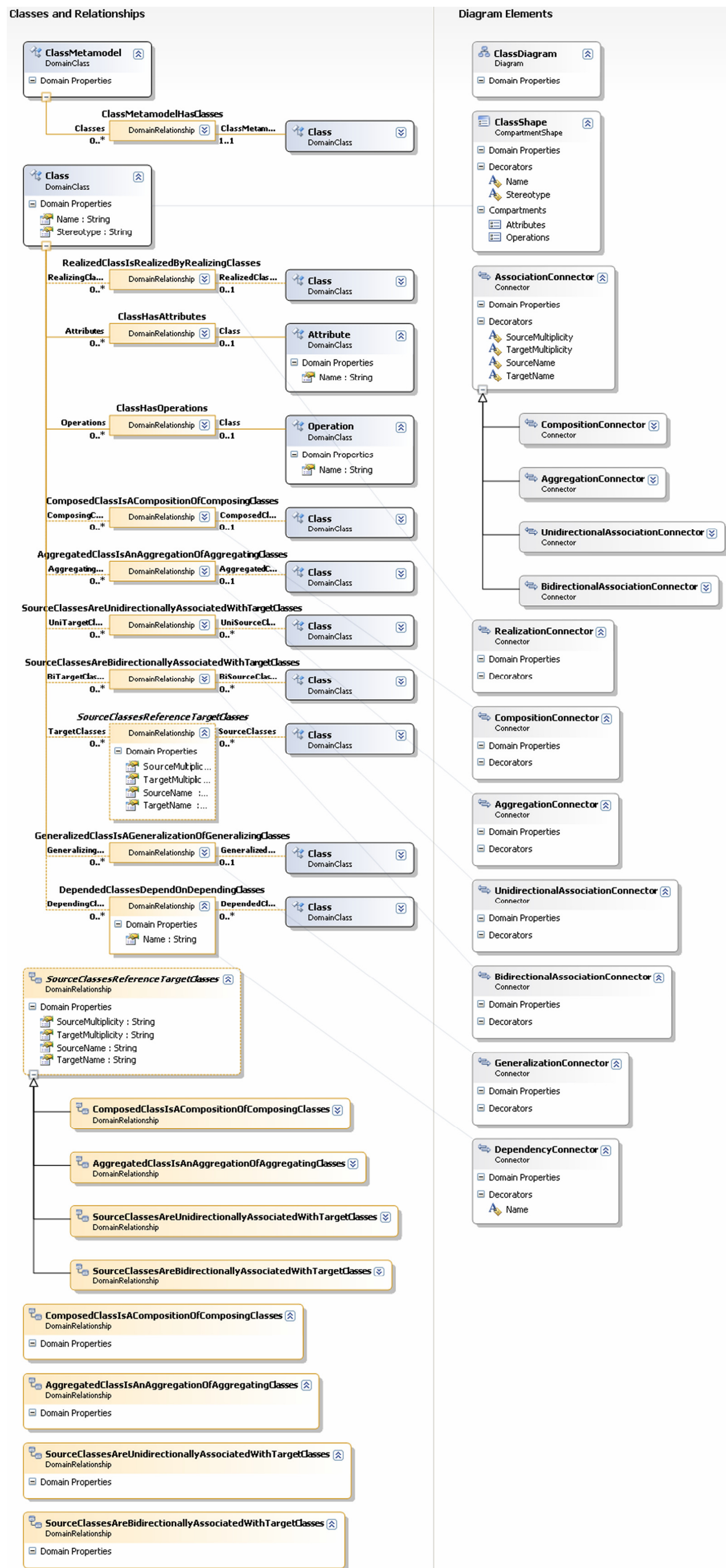

Figure 38 – Class metamodel that is part of the Primavera ERP metamodel for the sales domain

Some examples of elements from the metamodel in Figure 38 are instantiated in the class diagram illustrated in Figure 39, which are *Class*, *Attribute* and *SourceClassesAreBidirectionallyAssociatedWithTargetClasses*. The instances of *Class* are: *Artigo* (product), *FamiliaArtigos* (product family), *Fornecedor* (supplier), *Encomenda* (purchase), *CondicaoPagamento* (payment condition), *Vendedor* (sales person), *LinhaEncomenda* (purchase item), *Cliente* (customer) and *Zona* (zone). The instances of *Attribute* are each one of the attributes of each class, like *id* and *nome* (name) of the class *Cliente*. The instances of *SourceClassesAreBidirectionallyAssociatedWithTargetClasses* are most of the lines drawn between classes (bidirectional associations, or connectors in the terminology of DSL Tools).

The multiplicities associated with the ends of the instances of *SourceClassesAreBidirectionallyAssociatedWithTargetClasses* in Figure 39 are instances either of *SourceMultiplicity* or of *TargetMultiplicity*, decorators of the super connector *AssociationConnector*. This super connector allows that all associations that can use a source multiplicity, a target multiplicity, a source property and a target property inherit those properties from it and, so, the properties are defined only once.

«expressErpArtigo» and *«expressErpFornecedor»* are two examples of stereotypes illustrated in Figure 39, although written outside the class box. These stereotypes are instances of the decorator *Stereotype* of *ClassShape*. Furthermore, these stereotypes follow the stereotyping metamodelling hierarchy presented in section 3.4 of this dissertation.

No superclasses were used in the class diagram, therefore, the stereotypes are specific to a single application of the SPL (the free variant of the Primavera ERP in this case). The stereotypes which were used suit the purpose of configuring instances of UML concepts with concepts specific to the sales domain of the Primavera ERP in its free variant (which is the Primavera Express). The domain class *Class* is domain-specific because it has been combined with stereotypes.

Because the attributes (no operations were modelled) that affect more than one application of the ERP SPL mentioned in section 3.4 (which reside at the level of super domain concepts) were not modelled in the class diagram, it is not possible to distinguish between attributes specific to the free variant of the Primavera ERP and attributes the free variant shares with at least another variant (no superclasses were used in the diagram).

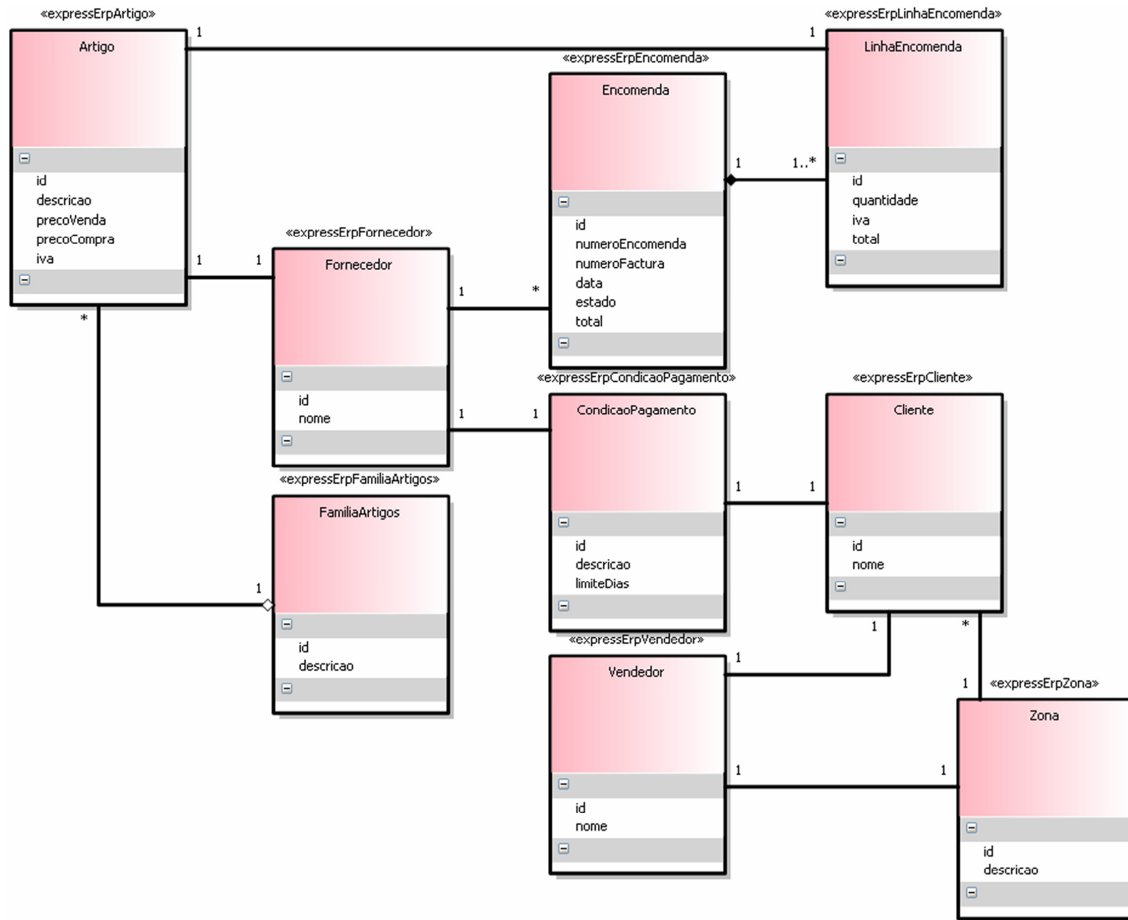

Figure 39 – Example of a class diagram in the context of sales

The stereotypes *«expressErpFornecedor»*, *«expressErpVendedor»* and *«expressErpCliente»* used in the class diagram are all instances of the meta-domain concept Actor already presented in section 3.4. All the other stereotypes are instances of another meta-domain concept that could be called Domain Object. The meta-domain concept Domain Object is intended to distinguish between the core business objects of the ERP SPL which are relative to an actor interacting with the system from all the others which are not.

Figure 40 shows the metamodel (or domain model) of the activity diagrams (like the one in Figure 41) conceived within the scope of the Primavera ERP metamodel for the sales domain.

The metamodel for the activity diagrams differs from the previous metamodels in the containment of a domain class called *ActivityPartition*. The activity partition has a swimlane shape associated called *ActivityPartitionShape*. The shape's decorator, *Name*, is the name of the partition.

A domain relationship exists in Figure 40 between the domain class *ActivityPartition* and the domain class *ActivityDiagramElement* called *ActivityPartitionHasActivityDiagramElements*. This relationship means that an activity partition may contain activity diagram elements, like decision or merge nodes (represented in the metamodel by the domain class *DecisionMergeNode*), fork or join nodes (represented in the metamodel by the domain class *ForkJoinNode*), actions (represented in the metamodel by the domain class *Action*), initial nodes (represented in the metamodel by the domain class *InitialNode*) and final nodes (represented in the metamodel by the domain class *FinalNode*). All these activity diagram elements have a generalization relationship with the domain class *ActivityDiagramElement*, which means that this domain class is abstract. The multiplicity of 0..* on the left side of the relationship *ActivityPartitionHasActivityDiagramElements*, associated with the domain property *ActivityDiagramElements*, means that the activity partition may have zero or more activity diagram elements.

The metamodel in Figure 40 presents a new aspect, which is a domain relationship, called *ControlFlow*, from the domain class, called *ActivityDiagramElement*, to itself. In the model, like the one in Figure 41, this domain relationship means that activity diagram elements can be connected to each other. An activity diagram element can be any of the specialized domain classes we can see in Figure 40, like *DecisionMergeNode*, *ForkJoinNode*, *Action*, *InitialNode* and *FinalNode*.

In Figure 40 the domain relationship *ControlFlow* is associated with the domain property *Condition*, which expresses the condition associated with the control flow instance in the model, as its name suggests.

Figure 41 depicts an activity diagram in the context of the Primavera ERP sales domain. The diagram models the operations the ERP processes for the use case *vender artigo* (sell product).

An operation is an action in the activity diagram. Each one of the orange boxes in Figure 41 is an instance of the domain class *Action*. The first element in the diagram is a black circle and an instance of *InitialNode*. The last element in the diagram, another black circle with a black circular line around it, is an instance of *FinalNode*. Each one of the arrows is an instance of *ControlFlow*. The outgoing arrows from the instances of the domain class *DecisionMergeNode*, the blue lozenges, have all an associated condition which is an instance of *Condition*, the domain property of the domain relationship *ControlFlow*.

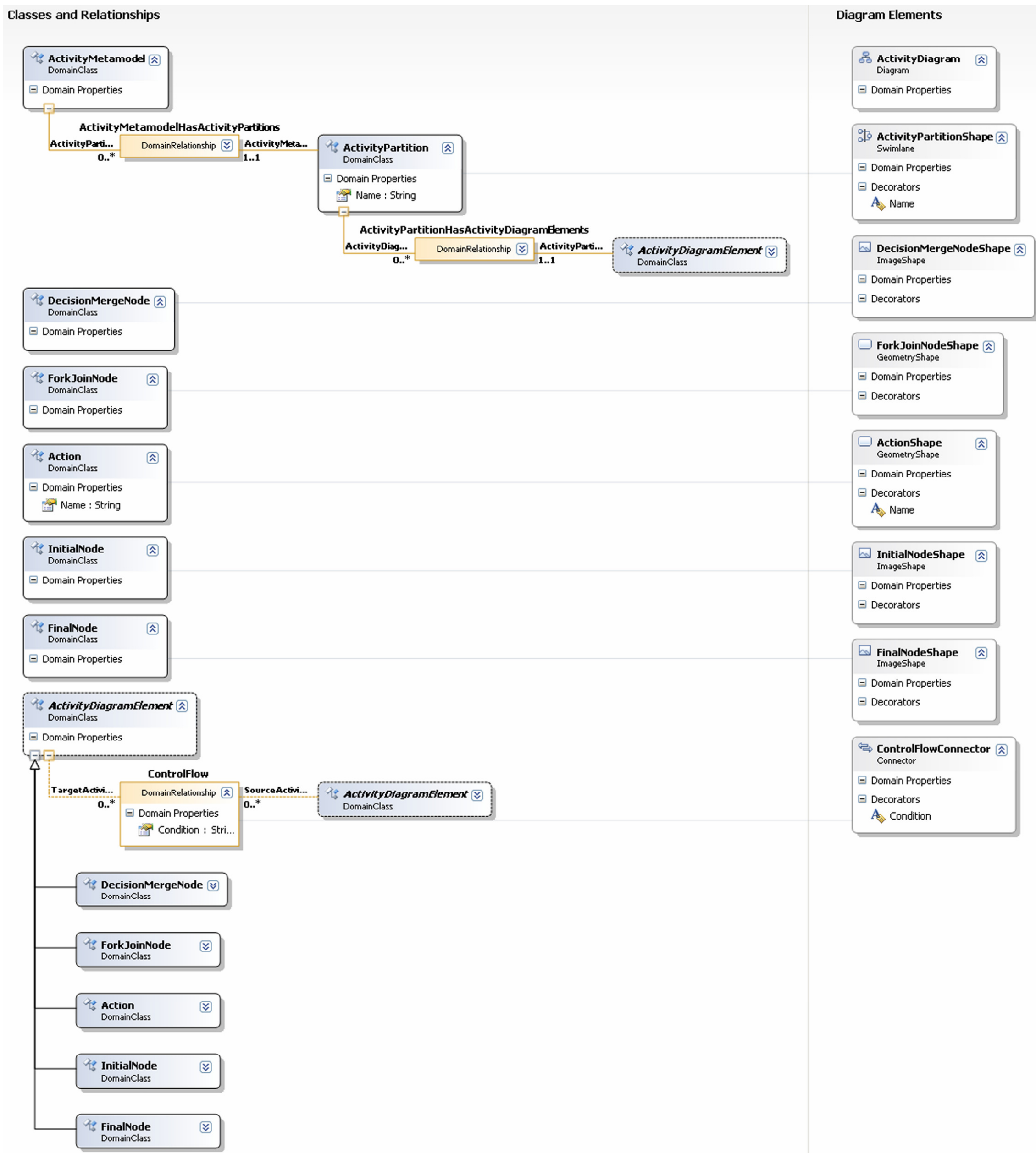

Figure 40 – Activity metamodel that is part of the Primavera ERP metamodel for the sales domain

The actors' names in Figure 41 correspond to domain concepts like those mentioned in section 3.4. Stereotypes could have been used. Some other domain concepts were used in the activity diagram, like *artigo* (product), *encomenda* (order), *conta corrente* (current account) and stock.

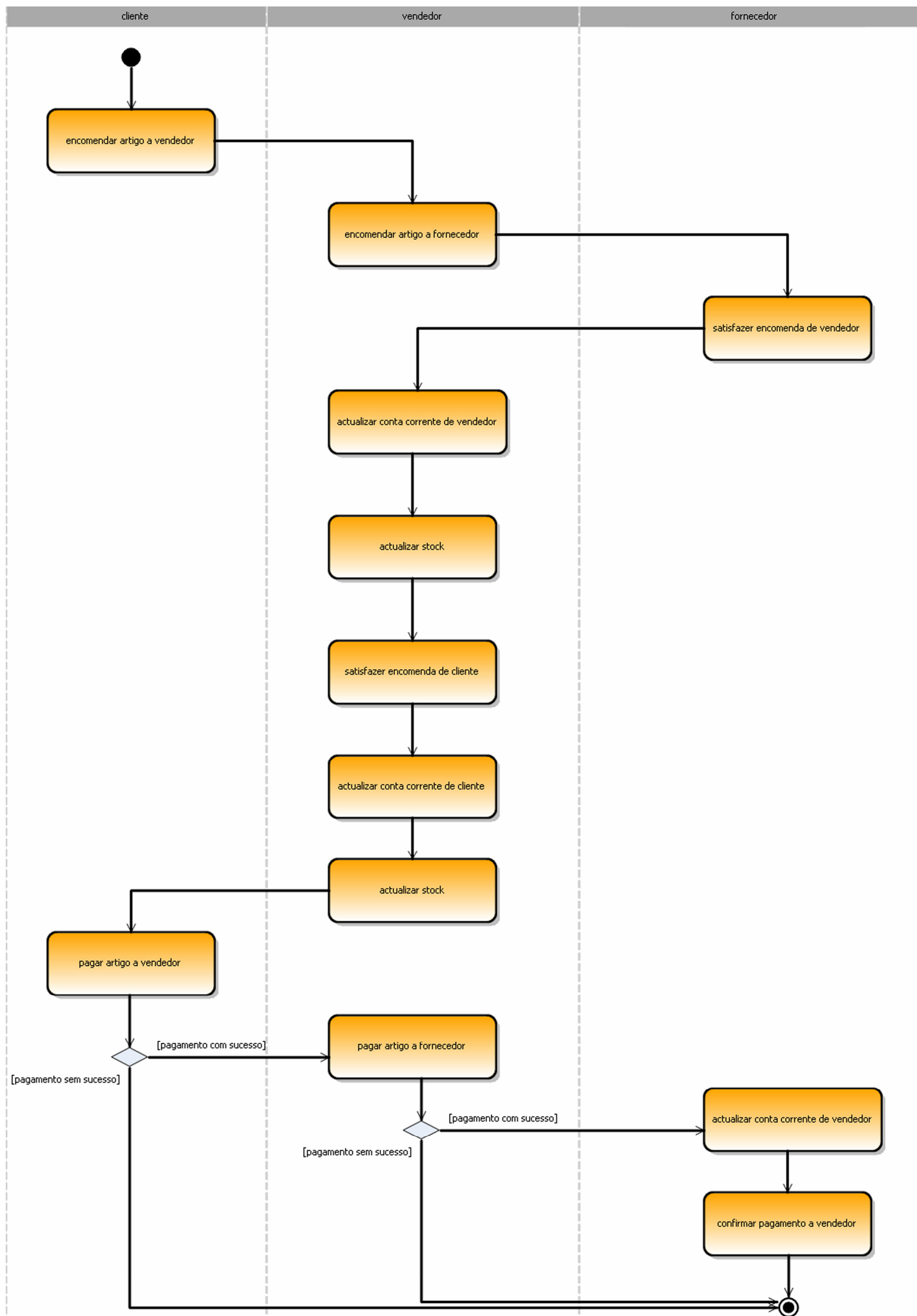

Figure 41 – Example of an activity diagram (sell product) in the context of sales

All the diagram elements in Figure 41 are organized inside partitions that are instances of *ActivityPartition*. There are three of them: *cliente* (customer), *vendedor* (sales person) and *fornecedor* (supplier). Each one of these names in the diagram is an instance of the domain property *Name* of the domain class *ActivityPartition* and represents an actor who interacts with the ERP. Each one of the actors assigned to each one of the partitions triggers the actions (or operations) inside that partition.

Figure 42 shows the metamodel (or domain model) of the state machine diagrams (like the one in Figure 43) conceived within the scope of the Primavera ERP metamodel for the sales domain.

The metamodel in Figure 42 presents a domain relationship, called *Transition*, from the domain class, called *StateMachineDiagramElement*, to itself. In the model, like the one in Figure 43, this domain relationship means that state machine diagram elements can be connected to each other.

A state machine diagram element can be any of the specialized domain classes we can see in Figure 42, like *FinalState*, *State*, *ChoicePseudostate* and *InitialPseudostate*. Associated with the domain relationship *Transition* is the domain property *GuardCondition*, which expresses the guard condition triggering the transition, as its name suggests.

Figure 43 depicts a state machine diagram in the context of the Primavera ERP sales domain. It represents the states of the object product during its lifecycle.

Each one of the yellow boxes in the state machine diagram is an instance of the domain class *State*. The names of the states, like *encomendado a vendedor* (ordered to sales person) or *encomendado a fornecedor* (ordered to supplier), are instances of the domain property *Name* of the domain class *State*. The black circle initializing the diagram is an instance of the domain class *InitialPseudostate*. The other black circle with a black circular line around it finalizing the diagram is an instance of *FinalState*. The blue lozenges are instances of the domain class *ChoicePseudostate*. The arrows connecting them are instances of *Transition* and are decorated with the guard condition, instance of the domain property *GuardCondition*.

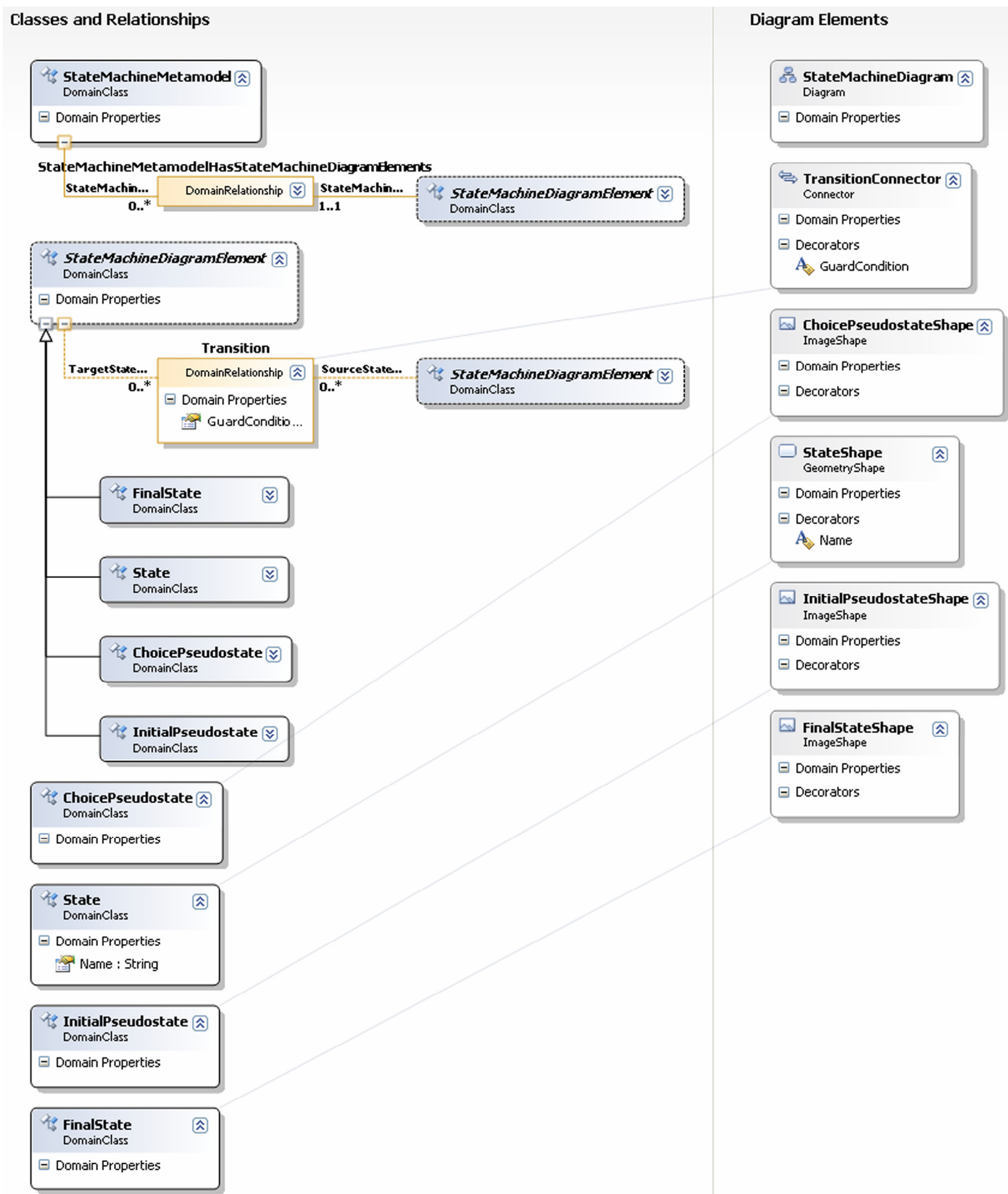

Figure 42 – State machine metamodel that is part of the Primavera ERP metamodel for the sales domain

Some domain concepts, like those mentioned in section 3.4, were used in the state diagram, like *vendedor* (sales person), *encomenda* (order), *fornecedor* (supplier), *pagamento* (payment) and *facturado* (from invoice). Stereotypes could have been used.

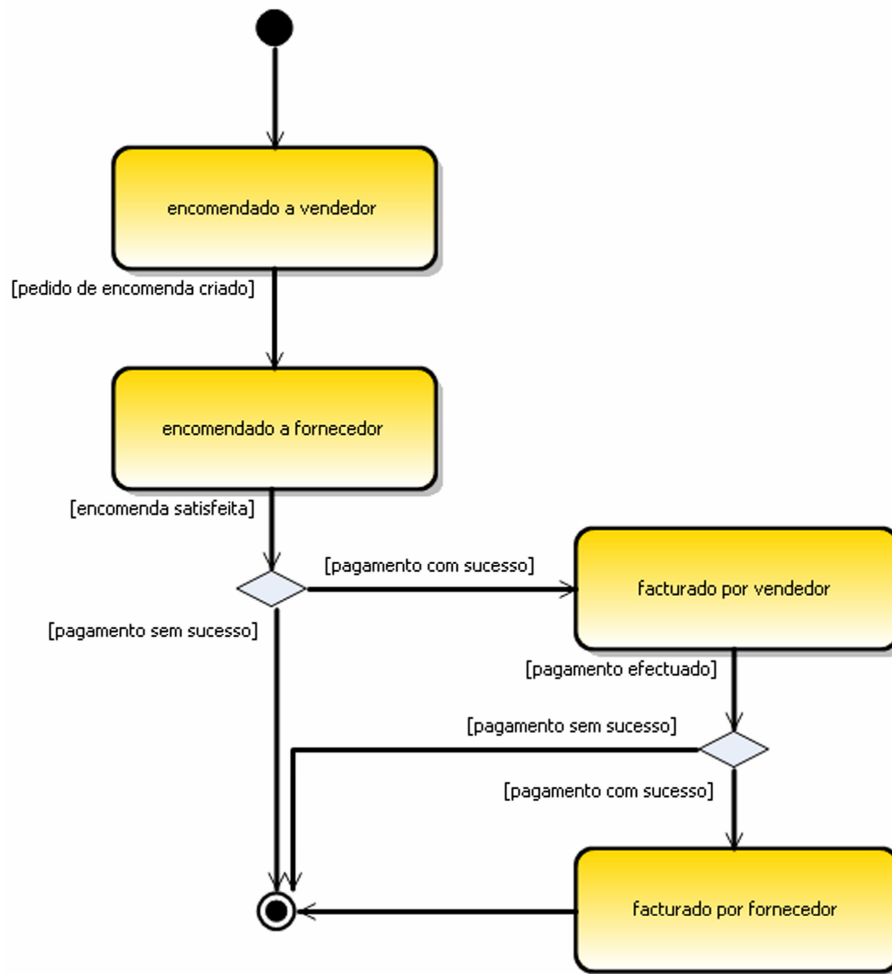

Figure 43 – Example of a state machine diagram (product) in the context of sales

4.4. Discussing the Undertaken Metamodelling Approach and the Conceived DSLs

The technical support for the conception of the Primavera ERP metamodel for the sales domain is clearly divided into two environments: a metamodelling environment and a modelling environment. The first one is used to produce the metamodels and the second one to produce the models. The first one is a tool to reason about the problem domain whereas the second one is a tool to reason about the solution domain, in concordance with what Brown*, et al.* [2006] stated. As long as the domain engineer is the domain expert, he should be the one conceiving the metamodels, since these artefacts establish the high-level domain concepts which concretize the problem he studies. By metamodelling the problem, the domain engineer makes available the metaconcepts that are going to be instantiated by the software engineer when designing the solution for that problem. This division of roles is depicted in Figure 44 (the figure includes the role of the software developer, the professional responsible for implementing the DSL). Actually, the software engineer is able to design the solution because he is experienced in the domain, yet, he does not have the experience expected from

a domain engineer working on the same domain of knowledge. Also, the software engineer knows better the ways to implement the solution to the problem using the target platform (in the case of Primavera ERP, the target platform is the .NET platform) than the domain engineer does. Hence, and following the reasoning of Brown*, et al.* [2006] of having different models suiting different purposes, on one hand, the metamodels the domain engineer creates in the metamodelling environment called *Designer* are used to establish a clear understanding of the problem. On the other hand, the models the software engineer creates in the modelling environment called the *Experimental Designer* by means of the DSL are used to communicate a clear vision of the solution to that problem. Again as Brown*, et al.* [2006] defended, DSL Tools allow the existence of models suited to the purpose of generating low level implementations from high level models: right from the metamodels, by just performing the *Transform All Templates* action, an implementation of the structure just conceived with the metamodel in the platform .NET is automatically generated.

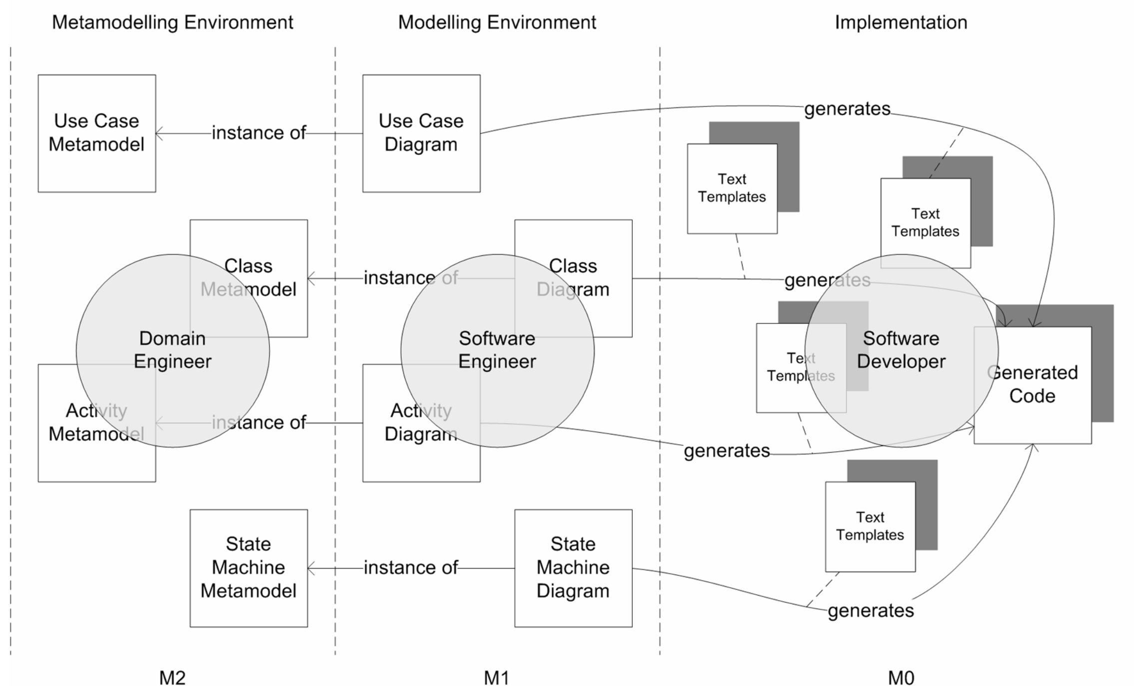

Figure 44 – Division of roles along the layered modelling architecture of Microsoft DSL Tools

In fact, the implementations automatically available from metamodels through DSL Tools are usable but still they could be more precise when it comes to the design of the application, in this case an ERP software solution. This fact is due to the abstraction level of metamodels. In terms of closeness to the platform, metamodels are above the abstraction level of models. Models are situated in the design phase instead of the analysis phase as the metamodels are, so, they are closer to the implementation than metamodels are. But as

Sendall and Kozaczynski [2003] argued, the Primavera ERP metamodel for the sales domain is also organized horizontally since different aspects of the system are expressed in different views of it.

Models can be either code-based or visual. If both coexist, then, abstraction levels affect both visual and code-based models, as in the case of the solution to the problem in discussion in this dissertation. A PIM exists along with a PSM at distinct levels of abstraction concerning closeness to target platform: the PIM is the metamodel at the highest level of abstraction or the model at the intermediate level of abstraction, whereas the PSM is the code generated out of the metamodel or the model and is at the lowest level of abstraction. The case in which the code is generated from the model can be seen in Figure 45. The generated code is at the lowest level of abstraction.

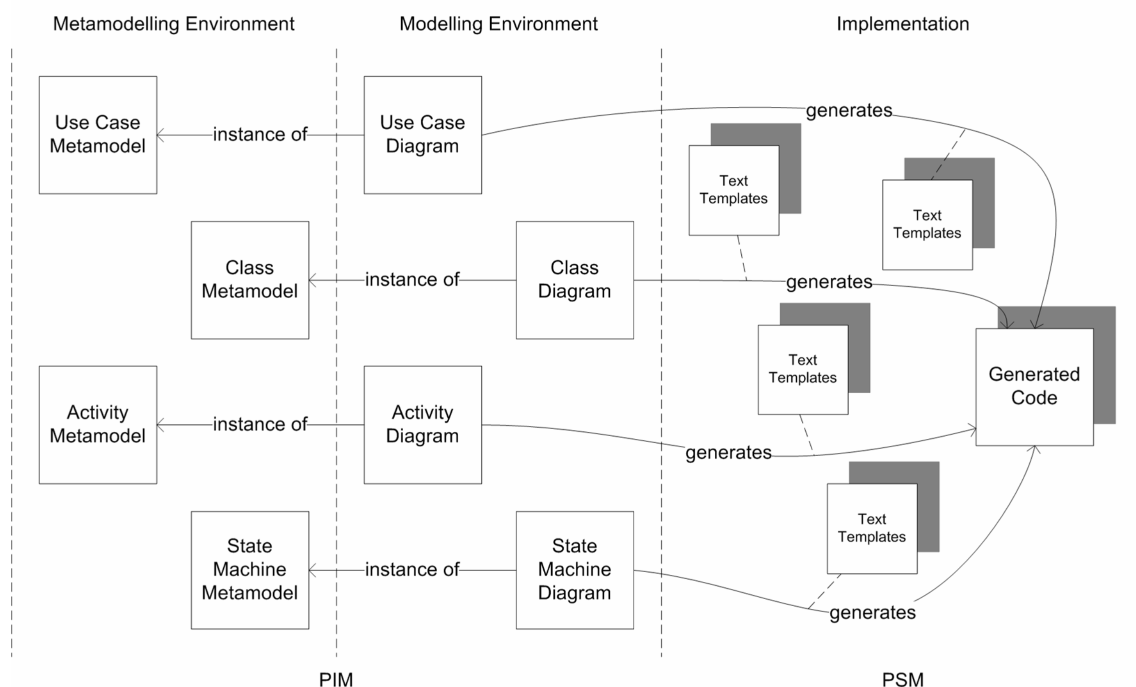

Figure 45 – PIM and PSM levels across the layered modelling architecture of Microsoft DSL Tools

MDD is concerned with reducing the impact of changes on software artefacts, as Atkinson and Kühne [2003] defended. If during the implementation of the software solution the need to add a new class to the solution is identified, what is needed is just to go back to the design phase, add that new class to the model and re-generate the code (if no impacts at the metamodel level are identified as well). The only thing left to do, then, will be to implement the logic, or internal behaviour, associated with that class. So, when a change needs to be done in the software solution, the new structure of the solution is assured to be implemented rapidly, which reduces dramatically the impact on development times.

Atkinson and Kühne [2003] stated that an MDD infrastructure shall provide a shared understanding of notation for creating models. This is precisely what the metamodels conceived with DSL Tools and the *Designer* environment make available: metamodels establish the entire notation available in the *Experimental Designer*; this notation is UML-based, therefore, its understanding is closer to the understanding of UML owing to the fact that it is a standard (that is the reason why a UML-based notation has been considered).

Having a metamodel and domain concepts available for a specific domain is of extreme relevance. They are the language basis for domain engineers and software engineers to communicate about the domain. The metamodel exposed in this dissertation is domain-specific when the domain-specific stereotypes are considered. The models are domain-specific because they use the stereotypes or the domain concepts attached to the stereotypes. DSL's definitions (or metamodels) and models built with the DSLs are domain-specific, in this case specific of ERPs. The exposed metamodel (without the stereotypes) is the first exercise to be done when metamodelling becomes the approach to develop software solutions. After having a metamodel based on UML, as the one this dissertation presented, the next step is to specify the basic domain concepts at the metamodelling level (the stereotypes and associated domain concepts) so that model elements use or consider those concepts.

According to Atkinson and Kühne [2003] and to Harel and Rumpe [2004], the definition of abstract syntax, concrete syntax and semantics is part of language definition. Therefore, the metamodels conceived with DSL Tools define the abstract syntax of the languages, the DSLs, used to conceive the models, as well as they define the concrete syntax through the notation's definition associated with each domain model, or metamodel. This way models' syntax is defined through metamodelling, like Sendall and Kozaczynski [2003] suggested. The semantics is defined for every metamodel in section 3.3.

The metamodels are conceived in DSMEs, which are metamodelling environments where DSLs are specified and from where *domain-specific environments* (the DSDEs) are generated [Lédeczi*, et al.*, 2001; Sprinkle*, et al.*, 2001]. Every *Experimental Designer* generated with Microsoft DSL Tools is a DSDE. A DSDE corresponds to each environment where each model was created. These environments allowed specifying the sales domain of the Primavera ERP software solution through models (in different views), as stated before in this dissertation. As Lédeczi*, et al.* [2001] argued in their work previously mentioned, the DSL, defined with a metamodel, is the source from which the DSDE is composed, or built, and this

is exactly what happens when handling DSL Tools and, particularly, the *Transform All Templates* action.

As far as transformations are concerned, DSL Tools allow mapping models into code (in a GPL) by means of a text template language, according to the suggestions Bettin [2004], Brown*, et al.* [2005a] and Demir [2006] have made on code generation. These transformations are used to produce family members rapidly and with low costs. In a SF approach, DSLs' definitions, like the ones metamodelled and presented in this dissertation, specify the common features of the product family that could be created from them (whenever a product is generated from models in different views, a code generation takes place, so, more than one code generation takes place when creating the product family). The variabilities of the SPL members would be, then, defined in the models. With this approach, DSL's reuse power would be explored for a family of applications.

4.5. Conclusions

Considering the UML metamodel's syntax and semantics in the design of the Primavera ERP metamodel for the sales module presented in its structural view, behavioural view and view of external functionalities and instances is, in fact, a metamodel inspired by the UML metamodel, which makes of it more comfortable to be interpreted by most of the stakeholders of the software development project (like those involved in the interpretation of the Primavera ERP). The reason is due to UML being a standard for general systems modelling with worldwide impact.

Regarding the metamodelling approach mentioned previously in this dissertation, it is demanding to situate this work within the OMG modelling infrastructure and distinguish between the different levels assumed during this work. The Primavera ERP metamodel for the sales module (see Figure 36, Figure 38, Figure 40 and Figure 42) is situated at the level referenced as M2 in Figure 3, the metamodel's level. All models (see Figure 37, Figure 39, Figure 41 and Figure 43), built from the respective metamodels, are situated at level M1, the model's level. The types of elements from the DSL Tools metamodel, mentioned in section 4.3.1 of this dissertation, are situated at level M3 of OMG's modelling infrastructure, the meta-metamodel's level.

As to code generation, which was slightly focused in this dissertation, we illustrated that it is possible to automatically generate code in a GPL from the definition of a DSL. Nevertheless, it is necessary to complete that code with hand-coded business logic further on in the software development process, which is the part of the development that demands more

resources and compromises some of DSLs' advantages pointed out in chapter 2. However, there is the possibility of manually adding text templates to the project of the *Experimental Designer*, compiled from the definition of the DSL. With these text templates in the *Experimental Designer* it is possible to automatically generate code from the models built within the *Experimental Designer*. It is possible, for instance, to obtain database mapping directives besides the definition of classes in a GPL.

The great effort a tool like Microsoft DSL Tools requires is in the conception of a metamodelling approach, like the focus of this work revealed. If this approach fails, all activities in the development of the software product that derive from the approach will fail as well. This represents added costs to the organization. It is at this point that organizations must bet before concentrating efforts in code generation.

5. Conclusions

> The last chapter of this dissertation in concerned with resuming the work reported in the document and provide suggestions to continue developing that work.

5.1. Results Analysis

The most embracing Software Engineering approach mentioned in this dissertation is MDD. Its main artefacts are models, which, in the case of Domain Engineering models, are used to reason about problem and solution domains for a particular knowledge area. But they suit more purposes, like to determine a high level artefact from which implementations can be generated. Abstraction plays a very important role in the implementation of software solutions. The way from high level artefacts to low level ones comprises some levels of abstraction and different types of abstraction. Closeness to platform and refinement, or detail, are the two types of abstraction. One of MDD's approaches is MDA. MDA is mainly about PIMs and PSMs, so, the major abstraction type at stake here is closeness to platform. MDA has the generation of code, the use of standards, a higher level of abstraction, visual modelling and metamodelling, besides the question of proximity to the target implementation platform, as its main requirements.

A big advantage of developing software with models is that a change in code can be rapidly done and is not so error prone as performing the change by hand. Models are also a

precious means that software developers and domain engineers can use to communicate. But MDD needs a set of requirements to take place. A standardized notation must be available to the conception of models. Models must be easily interchanged between tools. Finally, transformations between models, visual or code-based, must be possible.

The Four-Layer Architecture of UML is an important framework to be used by metamodelling approaches. The higher layers are MOF and UML concepts. These layers are concerned with language definition and are the metamodelling layers. MOF is equivalent to the meta-metamodel level and UML concepts to the metamodel level.

When defining a language it is fundamental to define its syntax and semantics. As to syntax, it can be abstract or concrete. Abstract syntax is expressed with metamodels and concrete syntax is the notation. Semantics is the textual description of a model's meaning.

Two kinds of environments were used in this dissertation. DSDEs are one of those types. This kind of environment allows specifying a software solution with models but within a specific domain context. These models, built with DSLs, are conceived during the design phase of the development of a domain-specific software solution, like an ERP. DSDEs can be partially composed from metamodels, which are no more than diagrams created in a DSME. Metamodelling languages allow specifying domain-specific modelling languages. Metamodels' syntax and semantics must be carefully handled.

Models may be subject of several and different transformations. The last kind of transformation to be performed is code generation. Transformations may be due to software evolution, or software maintenance. If requirements change, if external components change, if errors occur, software evolution must take place. Also if syntax and semantics change, if design patterns are applied, if SPLs are affected by the addition or retreat of components, software evolution happens. DSLs are affected by the changes software evolution brings with it.

Some software development methodologies were presented in this dissertation. Two of them are about SPL development. One of those is FAST and the other one is SF, which includes DSLs. FAST is about the development of an application engineering environment used to produce family members rapidly and with reduced costs. This kind of environment provides the production of SPL members more consistent with each other, as well as increased productivity levels, higher quality of code and reduced maintenance costs. SF' core is problem domain knowledge. Specialization in a particular domain is extremely significant. The distinction between commonalities and variabilities is also extremely important in this

methodology. Metamodels specify the commonalities of a SPL and models specify the variabilities of SPL members. DSLs play a vital role in determining the variabilities of SPL members. DSLs are defined through metamodels and visual constructs, in case of DSVLs (they are domain-specific embedded languages implemented through GPLs' constructs). DSLs have advantages, like higher quality of code (less error prone code), higher productivity levels, reduction of maintenance costs, closeness between design and analysis phases, and better understanding of software solutions by domain experts.

The development cycle of metamodelling environments is introduced in this dissertation. This cycle is particularly useful to organize the development of DSMEs around responsibilities (software developer's, domain engineer's and software engineer's). It follows the Four-Layer Architecture of UML in terms of levels. All the process is organized around models, as appropriate of an MDD approach. The software engineer is the bridge between the domain engineer and the software developer.

Microsoft DSL Tools combine DSMEs and DSDEs in order to, respectively, define modelling languages and use them to model domain-specific software solutions. Code generation is another feature supported by this tool.

The problem analysed in this dissertation is related to a part of the Primavera ERP software solution, a part of its sales domain. In order to define the DSLs previously reported in this dissertation, UML syntax was explored in section 3.3. UML syntax was considered during the DSLs' definition process. Each set of concepts from the UML metamodel concerning a specific view (use cases, classes, activities and state machines) is expressed in the corresponding metamodel built within DSL Tools. A DSL is presented for each one of the four treated views. During the conception of the four models exemplified, some of the Primavera ERP's business objects and interfaces were considered, as well as the UML semantics also explored in section 3.3 of this dissertation.

The DSL's definition comprises element types (domain classes) and relationship types (domain relationships), as well as it comprises stereotypes and domain concepts. The first two are defined in a domain model within DSL Tools. Notation is also defined along with the domain model. The stereotypes and the domain concepts are defined during the same stage in which the domain classes, the domain relationships and the notation are defined, which is the metamodelling stage.

When working with DSL Tools a major distinction exists between domain models, or metamodels, and models. The first ones are conceived within the DSMEs, whereas the

second ones are conceived within the DSDEs. The metamodels conceived within the DSME is possible because the metamodel of the language available to build metamodels in the DSME, or the DSL's meta-metamodel, is defined. It is of extreme significance to map some concepts of the UML metamodel into the corresponding concepts of the DSL Tools metamodel in order to define the notation to be used by the model elements in the DSDEs. After this mapping, it becomes clear which domain model's metamodel elements to use associated with which domain class or domain relationship.

The exercise with domain models exposed in this dissertation is the first step when it comes to incorporate metamodelling into a project of software development and it is relevant to perform this first exercise considering UML. UML-based domain models can be understood in closer way to the way UML is understood as a standard. Stereotypes follow this exercise. The stereotyping metamodelling hierarchy presented in this dissertation represents an approach to configuring instances of UML concepts (or instances of the above mentioned domain classes and domain relationships) with domain-specific concepts, as well as it defines the domain concepts which can be used in the models designed with the previously defined DSLs. Using these stereotypes and/or domain concepts defined for a part of the sales module of the Primavera ERP, along with the UML-based domain models is adopting a domain-specific approach (at the levels of metamodelling and modelling) tailored to the sales domain of the Primavera ERP. In the end, a syntactic base for the context of the Primavera ERP sales domain is established with the DSLs, the stereotypes and the domain concepts.

The metamodelling approach this dissertation proposed, which was used to define the DSLs within DSL Tools, is composed by the delineation of UML-based syntax, both abstract (the domain models conceived within DSL Tools) and concrete (the mapping between some UML concepts and the corresponding DSL Tools' concept and the notation's definition done in the DSMEs) syntaxes. The stereotypes and the domain concepts defined at the metamodelling level are also part of the metamodelling approach.

5.2. Future Work

As previously mentioned in this dissertation, a metamodelling approach is not complete unless framed by a software development methodology. Although, and very important, a metamodelling approach can consider UML so that every stakeholder is highly capable of understanding what is intended to be clarified with the metamodels, that is not enough. The probability of having low process effectiveness is high, if metamodels are

conceived with no software development methodology involving them, a methodology imposing clear rules to the metamodelling of the domain. The first consequences in the development of the software solution are going to be excessive returns to the analysis and design phases to change the metamodel(s) or model(s), respectively. These changes could have been avoided, if already inserted in the metamodel(s) or model(s) early in the software development phases, causing higher process effectiveness.

It is also important to define stereotypes affecting the different Primavera's products and using them when creating the DSLs. In this dissertation only one of those products, the Primavera Express, was considered, but all of them need to be considered as well.

A software development methodology, like 4SRS, can be applied to metamodelling approaches like the one this dissertation presented. This methodology, as stated before, is targeted at transforming requirements specifications in the form of use case diagrams and respective textual descriptions into the logical architecture of the system. The domain models can be refined from this logical architecture in order to incorporate at the metamodelling level the domain concepts of the system which may be derived from the logical architecture. Still, the approach of conceiving UML-based metamodels and models must not be discarded, as it is a part of the methodology, a kind of pre-metamodelling and pre-modelling phase.