Model-Driven Domain Analysis and Software Development:
Architectures and Functions

Janis Osis
Riga Technical University, Latvia

Erika Asnina
Riga Technical University, Latvia

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE
Hershey · New York

Director of Editorial Content:	Kristin Klinger
Director of Book Publications:	Julia Mosemann
Acquisitions Editor:	Lindsay Johnston
Development Editor:	Joel Gamon
Typesetter:	Keith Glazewski
Production Editor:	Jamie Snavely
Cover Design:	Lisa Tosheff

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Model-driven domain analysis and software development : architectures and functions / Janis Osis and Erika Asnina, editors.
p. cm.
Includes bibliographical references and index.
Summary: "This book displays how to effectively map and respond to the real-world challenges and purposes which software must solve, covering domains such as mechatronic, embedded and high risk systems, where failure could cost human lives"--Provided by publisher.
ISBN 978-1-61692-874-2 (hardcover) -- ISBN 978-1-61692-876-6 (ebook) 1. Automatic control. 2. Computer simulation. 3. Computer software--Development. I. Osis, Janis, 1929- II. Asnina, Erika, 1978-
TJ213.M5373 2010
005.1--dc22
2010043567

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 14
Systematic Use of Software Development Patterns through a Multilevel and Multistage Classification

Sofia Azevedo
Universidade do Minho, Portugal

Ricardo J. Machado
Universidade do Minho, Portugal

Alexandre Bragança
Instituto Superior de Engenharia do Porto, Portugal

Hugo Ribeiro
Primavera Business Software Solutions, Portugal

ABSTRACT

Software patterns are reusable solutions to problems that occur often throughout the software development process. This chapter formally states which sort of software patterns shall be used in which particular moment of the software development process and in the context of which Software Engineering professionals, technologies and methodologies. The way to do that is to classify those patterns according to the proposed multilevel and multistage pattern classification based on the software development process. The classification is based on the OMG modeling infrastructure or Four-Layer Architecture and also on the RUP (Rational Unified Process). It considers that patterns can be represented at different levels of the OMG modeling infrastructure and that representing patterns as metamodels is a way of turning the decisions on their application more objective. Classifying patterns according to the proposed pattern classification allows for the preservation of the original advantages of those patterns and avoids that the patterns from a specific category are handled by the inadequate professionals, technologies and methodologies. The chapter illustrates the proposed approach with the classification of some patterns.

DOI: 10.4018/978-1-61692-874-2.ch014

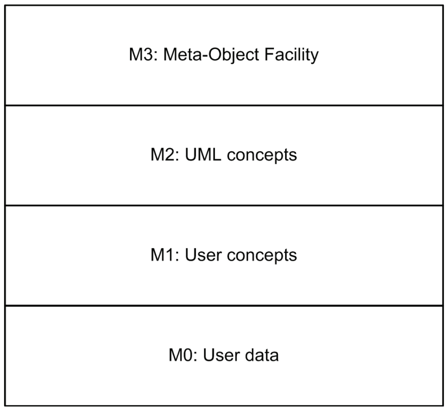
INTRODUCTION

In the context of software development, patterns are provided as reusable solutions to recurrent problems. In other words, software patterns are reusable solutions to problems that occur often throughout the software development process. Pattern classifications emerged as a way to organize the many patterns that have been synthesized. *Pattern classification* is the activity of organizing patterns into groups of patterns that share a common set of characteristics. The simple fact of organizing patterns into classifications is a way of building a stronger knowledge on patterns, which allows understanding their purpose, the relations between them and the best moments for their adoption (Gamma, Helm, Johnson, & Vlissides, 1995).

Despite their use within the software development process, the use of patterns may not be systematic. In the context of this chapter, the systematic use of software development patterns means that decisions on the application of patterns are less subjective and more objective. Besides that, a lot of pattern classifications were conceived until the present day, yet none of them formally stated which sort of patterns shall be used in which particular moment of the software development process. This chapter will provide for specific directives on how to systematically adopt patterns within a multilevel and multistage software development process. A multilevel and multistage classification of patterns will be the foundation of such systematic use of patterns.

A multistage software development process can be defined as a software development process composed of some stages organized in a consecutive temporal order. Each stage is separated from the contiguous ones by well defined borders. Moreover each particular stage is composed of a flow of well defined activities. Each stage's activities are conducted by specific professionals, using specific technologies (frameworks, languages, tools), under the directives of specific methodologies (processes, notations and methods) to achieve specific goals. Borders are well defined if the shift in the professionals, technologies, methodologies and goals that takes place when moving from one stage to another is identified in terms of the development process. A multilevel software development process can be defined as a software development process concerned with the levels of abstraction in which the different artifacts involved in the development of software are handled. In the context of this chapter, those levels are the levels of the OMG (OMG, 2009a) modeling infrastructure or Four-Layer Architecture (Atkinson & Kühne, 2003), depicted in Figure 1. The OMG modeling infrastructure comprises a hierarchy of model levels just in compliance with the foundations of MDD (Model-Driven Development) (Atkinson & Kühne, 2003). Each model in the Four-Layer Architecture (except for the one at the highest level) is an instance of the one at the higher level. The first level (*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 *language concepts models*. These are models of models and so are called metamodels. A metamodel is a model of a modeling language. It is also a model whose

Figure 1. The OMG modeling infrastructure or Four-Layer Architecture



elements are types in another model. An example of a metamodel is the UML (Unified Modeling Language) metamodel (OMG, 2009b). 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 *language concepts metamodels* are at the highest level of the modeling infrastructure. 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.

Patterns are provided by pattern catalogues such as (Adams, Koushik, Vasudeva, & Galambos, 2001; Beck, 2008; Buschmann, Meunier, Rohnert, Sornmerlad, & Stal, 1996; Eriksson & Penker, 2000; Fowler, 1997, 2003b; Gamma, Helm, Johnson, & Vlissides, 1995; Larman, 2001; Pree, 1995). Pattern languages are more than pattern catalogues (collections of patterns). A pattern language is composed of patterns for a particular (small and well-known) domain. Those patterns must cover the development of software systems down to their implementation. A pattern language must also determine the relationships between the patterns the language is composed of. The language's patterns are its vocabulary, and the rules for their implementation and combination are its grammar (Buschmann, Meunier, Rohnert, Sornmerlad, & Stal, 1996).

The *adoption* of a pattern (pattern adoption) is composed by the set of activities that consist of using the pattern somehow when producing software artifacts. Namely those activities are: (1) pattern interpretation; (2) pattern adaptation; and (3) pattern application. Patterns have to be interpreted in order to be applied. For the reason that usually patterns are not documented by those who apply them, they have to be interpreted prior to their application. The *interpretation* of a pattern is the activity that consists of reading the pattern from the pattern catalogue and reasoning about the solution the pattern is proposing for that problem in that given context. Following the interpretation activity, the adoption process may require the patterns to be adapted somehow (Beck, 2008; Fowler, 2003a). The *adaptation* of a pattern is the activity of modifying the pattern from the catalogue without corrupting it (corrupting the pattern includes corrupting the pattern's semantics and the pattern's abstract syntax). Finally the *application* of a pattern is its actual use in the development of software, whether to develop software products or families of software products, or to inspire the conception of design artifacts since some patterns are not identifiable in the source code as they are not meant to give origin to code directly (Soukup, 1995).

Habitually pattern catalogues represent patterns at the M1-level of the OMG modeling infrastructure or Four-Layer Architecture. We consider that leveraging patterns to the M2-level is a way of turning the decisions on their application more objective as well as of reducing the misinterpretation of patterns from catalogues and the corruption of patterns during the pattern adaptation process. Misinterpretation and corruption of patterns can lead to the irremediable loss of the advantages of adopting those patterns. Considering the OMG modeling infrastructure as a multilevel architecture, *multilevel instantiation* (or the instantiation of M2-level patterns at the M1-level) shall occur during the adoption of patterns.

This chapter is an original contribution to the improvement of the software products' quality given that it provides for some directives on how to adopt software patterns in such a way that the original advantages of the adopted pattern are preserved. The originality of the contribution is due to the novelty character of the pattern classification, which relies on the fact that it is based on the software development process. The classification we propose represents a benefit in terms of the process of developing software as it allows knowing (by classifying the patterns according to it) in which moment of the software development process to use the patterns and in the context of which Software Engineering professionals, technologies and methodologies. This chapter contributes for MDD since it addresses the OMG modeling infrastructure through the multi-level character of the proposed classification. The classification considers that patterns can be represented at different levels of the OMG modeling infrastructure, which influences their interpretation. The usefulness of a multilevel and multistage pattern classification resides in avoiding that the patterns from a specific category are handled by the inadequate professionals, technologies and methodologies. By classifying the patterns (in this case the software development patterns) we assure that the professionals with the right skills (who use the technologies and methodologies adequate to their profile) use the right pattern categories. For instance it would be inadequate for a product manager to use a pattern from the Gang of Four (GoF) book (Gamma, Helm, Johnson, & Vlissides, 1995). That would not produce the desired effects of using such kind of pattern.

This chapter is structured as follows: Section 2 affords a state-of-the-art that suits the purpose of substantiating the strength of our approach; Section 3 aims at clarifying the relation of patterns, pattern classifications and the proposed pattern classification with the theme of the book; Section 4 is devoted to exhibiting the proposed pattern classification in abstract terms before formalizing categories and positioning patterns at those categories; Section 5 is targeted at demonstrating the feasibility of the solution we are going to propose to the systematic use of software development patterns by using some concrete examples of patterns positioned at distinct categories of our classification to illustrate the different types of patterns we have formalized; finally Section 6 exposes some concluding remarks.

BACKGROUND

Typically patterns are adopted at later phases of the software development process. The analysis and design phases of software development are disregarded. Most of the times analysis and design decisions are not documented and that originates missing knowledge on how the transition from previous stages to the implementation stage was performed. Knowing design decisions without design documentation as a helper of this activity is only possible if those decisions can be transmitted by the people who know them. When talking about patterns, design decisions have to be perfectly known so that an activity of pattern discovery can be applied to a software solution with the purpose of discovering the original pattern (the pattern in the catalogue) from the implementation. If the original pattern is successfully reengineered from the implementation, then it means that most likely the advantages of the original pattern are present in that software solution. It is pertinent to understand how patterns from catalogues, after being interpreted, adapted and applied, can be constrained in such a way that the advantages enclosed in the solution each of those patterns proposes cannot be observed. Buschmann, *et al.* (Buschmann, Henney, & Schmidt, 2007b) referred that patterns may be implemented in many different ways; still patterns are not vague in the solution structure they propose. The diversity in the instantiations of a pattern is due to the specificity of the concrete problems being addressed. What must be assured is the "spirit of the pattern's message" as Buschmann, *et al.* called it. In the development of software it must be assured that not only the advantages of the original pattern are visible (directly or indirectly) in the software solution but also that patterns are adopted throughout all the process phases since patterns address all of them as we will be seeing in the next section of this chapter. Besides these two considerations it must be noted that the development of software is not performed exclusively based on patterns but it is a microprocess or nanoprocess when compared to the whole software development process as Buschmann, *et al.* stated.

Pattern classifications are useful for understanding pattern catalogues better and providing input for the discovery of new patterns that fit into the already existing pattern categories (Gamma, Helm, Johnson, & Vlissides, 1995). Patterns are classified into categories according to different classification criteria and are organized in pattern catalogues according to classification schemas that support the different classification criteria each particular schema contemplates. Classification schemas can be unidimensional or multidimensional depending on whether they obey to a single or more than one criterion. Throughout this chapter (due to simplification purposes) we are going to use the term pattern classification instead of the complete term pattern classification schema.

The pattern classifications of (Beck, 2008; Eriksson & Penker, 2000; Gamma, Helm, Johnson, & Vlissides, 1995; Pree, 1995; Tichy, 1997; Zimmer, 1995) have not been explicitly defined within a procedural referential, thus we are not able to know beforehand which software pattern shall be used at what moment during the process of developing software in general as well as in the context of which Software Engineering professionals, technologies and methodologies. These procedural concerns include also the adoption of a modeling infrastructure to prevent subjective pattern application decisions, and situations of misinterpretation and corruption of patterns from catalogues while interpreting and adapting the patterns respectively. At last the classifications we are going to present next have not elaborated on the nature of the domain to which patterns are most adequately applicable. Considering that nowadays families of software products are commonly developed with domain-specific artifacts, taking the adequacy of patterns to particular domain natures into account is relevant in order to choose between the patterns that are most applicable to a domain-specific software product or family of products.

The first pattern classification we mention is from the GoF (Gamma, Helm, Johnson, & Vlissides, 1995). They classified design patterns according to two criteria: purpose and scope. The purpose of a pattern states that pattern's function. According to the purpose, patterns can be *creational*, *structural* or *behavioral*. Creational patterns are concerned with the creation of objects. Structural patterns are targeted at the composition of classes or objects. Behavioral patterns have to do with the interaction between classes or objects and their responsibility's distribution. The scope of a pattern is its applicability either to classes or to objects. *Class patterns* are related to the relationships between classes. *Object patterns* are related to the relationships between objects. Despite the GoF's classification considering more than one criterion, it is not multidimensional as the criteria have not been combined to determine pattern categories. The GoF's classification is concerned with the function of the pattern (what the pattern does) and its applicability to low level implementation elements (how the pattern will be handled in the software construction moment). The classification does not refer to explicit procedural questions on the development of software with the use of patterns (when patterns shall be used, by whom, with what technologies and methodologies, and at which levels of abstraction) or to questions with the applicability of patterns to specific domain natures. The same is true for the classification we are going to mention next.

A classification of patterns according to their relationships was proposed by Zimmer (Zimmer, 1995). Zimmer classified the relationships into three categories: *X uses Y in its solution* (the solution of X contains the solution of Y), *X is similar to Y* (both patterns address a similar type of problem) and *X can be combined with Y* (both patterns can be combined, in spite of the solution of X not containing the solution of Y). This classification may give hints on the selection and composition of patterns, nevertheless it does not provide for directives on the nature of the domain the patterns are more adequate to, on the right moment to adopt the patterns, within which Software Engineering discipline's context and on how to respect a modeling infrastructure when adopting the patterns.

A classification of general-purpose design patterns (patterns traversal to all application domains) was proposed by Tichy in (Tichy, 1997). Tichy proposed nine categories to organize design patterns. The categories were determined based on the problems solved by the patterns. The proposed categories were *decoupling* (which has to do with the division of a software system into independent parts), *variant management* (which is associated with the management of commonalities among objects), *state handling* (which is the handling of objects' states) and others. Again, neither procedural concerns, nor concerns with the applicability of patterns to particular domain nature types were evidenced by this classification that relies on the types of problems patterns propose to solve.

The Pree's and the Beck's classifications we are going to expose next do not also evidence hints on which moments of the software development process to adopt patterns, in the context of which Software Engineering discipline, respecting a modeling infrastructure and the applicability of patterns to domain natures in particular.

Wolfgang Pree (Pree, 1995) categorized design patterns by distinguishing between the purpose of the design pattern approach and its notation. Notation can be informal textual notation (plain text description in a natural language), formal textual notation (like a programming language) or graphical notation (like class diagrams). Purpose expresses the goal a design pattern is pursuing. The *Components* category indicates that design patterns are concerned with the design of components rather than frameworks. The *Frameworks I* category indicates that design patterns are concerned with describing how to use a framework. The *Frameworks II* category indicates that design patterns represent reusable framework designs. Pree's classification scratches very superficially the question of modeling as it distinguishes between patterns represented with code (formal textual notation in the Pree's classification) and those represented with models (graphical notation in the Pree's classification) but it does not elaborate on how to work respecting different levels of abstraction throughout the process of developing software.

Kent Beck's (Beck, 2008) implementation patterns translate good Java programming practices whose adoption produces readable code. He claims these are patterns because they represent repeated decisions under repeated decision's constraints. Kent Beck's implementation patterns are divided into five categories: (1) class, with patterns describing how to create classes and how classes encode logic; (2) state, with patterns for storing and retrieving state; (3) behavior, with patterns for representing logic; (4) method, with patterns for writing methods (like method decomposition, method naming); and (5) collections, with patterns for using collections. Kent Beck claims his implementation patterns describe a style of programming. These implementation patterns address common problems of programming. For instance Kent Beck advises to use the pattern *Value Object* if the intention is to have an object that acts like a mathematical value, or the pattern *Initialization* for the proper initialization of variables, or the pattern *Exception* to express non-local exceptional flows appropriately, or the pattern *Method Visibility* to determine the visibility of methods while programming, or the pattern *Array* as the simplest and less flexible form of collection. Kent Beck uses Java in order to exemplify the pattern (as a different presentation of it) instead of a model or a structured text. Despite the programming practices having to be considered by the software development process, this classification does not care about the process of adopting patterns within the whole software development process.

Not only design patterns and implementation patterns are used when developing software. The classification of Eriksson and Penker (Eriksson & Penker, 2000) addresses business-level patterns like those we are going to mention just now. The *Core-Representation* pattern dictates how to model the core objects of a business (the *business objects* e.g. customer, product, order) and their representations (e.g. the representation of a business object within the information system may be a window or another graphical user interface element as the representation of a debt is an invoice and the representation of a country may be the country code). The *Document* pattern shows how to model documents (e.g. how to handle different versions and copies of a document). The *Geographic Location* pattern illustrates how to model addresses (which is of interest to mail-order companies, post offices, shipping companies). The *Organization and Party* pattern demonstrates how to model organizational charts. The *Product Data Management* pattern indicates the way to model the structure of the relationship between documents and products (the structure varies from one business to another). The *Thing Information* pattern (used in e-business systems) models the thing (resource in the business model) and the information about the thing (the information in the information system about that resource). The *Title-Item* pattern (used by stores and retail outlets) is to model items (e.g. a loan item) and their titles (e.g. a book title). The *Type-Object-Value* pattern (used by geographical systems) depicts how to model the relationship between a type (e.g. country), an object (e.g. Portugal) and a value (e.g. +351). Eriksson and Penker classified business-level patterns into three categories: *resource and rule patterns*, *goal patterns* and *process patterns*. The *resource and rule patterns* provide for guidelines on how to model the rules (used to define the structure of the resources and the relationships between them) and resources (people, material/information and products) from a business domain. The *goal patterns* are intimately related to goal modeling. The main idea is that the design and implementation of a system depends on the goals of the system (how it is used once built). At last the *process patterns* are related to process-oriented models (such as workflow models). Process patterns prescribe ways to achieve specific goals for a set of resources, obeying to specific rules that express possible resource states.

The classification we are going to mention next is elaborated on the software development phases. Siemens' (Buschmann, Meunier, Rohnert, Sornmerlad, & Stal, 1996) two-dimensional pattern classification (from the book "Pattern-Oriented Software Architecture" (POSA), volume 1, or just POSA 1) was defined with two classification criteria (*pattern categories* and *problem categories*). Every pattern is classified according to both criteria. The *pattern categories* determined were *architectural patterns*, *design patterns* and *idioms*. They are related to phases and activities in the software development process. Architectural patterns are used at early stages of software design, particularly in the structure definition of software solutions. Design patterns are applicable to former stages of software design, particularly to the refinement or detailing of what Buschmann, *et al.* call the *fundamental architecture of a software system*. Idioms are adequate to implementation stages, where software programs are written in specific languages. The *problem categories* determined were *from mud to structure*, *distributed systems*, *interactive systems*, *adaptable systems*, *structural decomposition*, *organization of work*, *access control*, *management*, *communication* and *resource handling*. As an example *Structural Decomposition patterns* support the decomposition of subsystems into cooperating parts and *Organization of Work patterns* support the definition of collaborations for the purpose of providing complex services. These categories express typical problems that arise in the development of software. Placing some patterns in a specific category is a useful activity since it allows eliciting related problems in software development. However this pattern classification does not address the analysis phases (business modeling and requirements) of the software development process as the multilevel and multistage pattern classification does.

The POSA 1 (Buschmann, Meunier, Rohnert, Sornmerlad, & Stal, 1996) and the POSA5 (Buschmann, Henney, & Schmidt, 2007b) are the most general POSA references. The POSA 2 (Schmidt, Stal, Rohnert, & Buschmann, 2000) contains a pattern language for concurrent and networked software systems. The POSA 3 (Kircher & Jain, 2004) contains a pattern language for resource management. The POSA 4 (Buschmann, Henney, & Schmidt, 2007a) contains a pattern language for distributed computing. As referred in the POSA 5 by its authors the classifications in the POSAs 2, 3 and 4 are intention-based, which is why they haven't been included in this chapter's literature review. This chapter is targeted at software development patterns in general, not intention-based software development patterns.

In the POSA 5 Buschmann, *et al.* reflect on the terminology used in the pattern classification in the POSA 1 and conclude that the pattern classification from the POSA 1 has terminology problems. The terms used to distinguish disjoint categories (*architectural patterns*, *design patterns* and *idioms*) actually do not refer to pretty disjoint categories. These authors refer that architectural activities and the application of *idioms* can also be considered design activities. They also refer that since the POSA 1 they have concluded that the term *design pattern* is to designate software development patterns in general and to distinguish them from patterns that have nothing to do with software. It does not mean that they have to do with design activities. For this reason they conclude that the term *design pattern* used in the pattern classification in the POSA 1 should have been replaced with some other name to refer to the GoF patterns. Concerning the *architectural patterns* Buschmann, *et al.* conclude that all patterns are architectural in nature, so there cannot be a category called *architectural patterns*. To Buschmann, *et al.* design is the activity of making decisions on the structure or behavior of a software system and architecture is about the most significant design decisions for a system (and not all design decisions). Therefore although all patterns are intrinsically architectural, not all of them are applicable to architectural activities. Concerning the *idioms*, Buschmann, *et al.* conclude that the term *idiom* has some ambiguity since sometimes it refers to a solution for a problem specific to a given programming language and some other times it refers to conventions for the use of a programming language. An *idiom* can even refer to both situations. Buschmann, *et al.* also conclude that *idioms* can refer to patterns used within the context of a specific domain, architectural partition or technology, thus they conclude that the term *idiom* should have been *programming language idiom* as a programming language is a specific solution domain. For instance the pattern *Iterator* is an *idiom* specific to C++ and Java, although it differs between these two specific languages.

Since all architecture is design (Clements et al., 2002) the consideration of Buschmann, *et al.* that there cannot be a pattern category for architectural patterns makes sense (they are patterns of design). However not all design is architecture (Clements et al., 2002), which means that a distinction between patterns that address architecture and patterns that address design has to be made. Architectures do not define implementations, they rather constrain downstream activities of design and implementation. The architecture defines the system structure. The software architect shall leave the implementation details veiled. Design patterns shall address details of implementation (like the GoF patterns do).

The matter with *idioms* that Buschmann, *et al.* mention in the POSA 5 has been solved by Kent Beck in (Beck, 2008). Kent Beck's implementation patterns express good programming practices (or the conventions for the use of programming languages). Kent Beck uses Java in order to exemplify his implementation patterns, which shall be applicable to other programming languages. Kent Beck's implementation patterns are not Java or other language-specific patterns that are just a different representation of design patterns (Grand, 2002; Stelting, 2002).

PATTERNS AND MODEL-DRIVEN SOFTWARE DEVELOPMENT

Atkinson and Kühne discuss the foundations of MDD in (Atkinson & Kühne, 2003). The goal of MDD is to raise the abstraction level at which software programs are written by reducing the software development effort needed to produce a software product or set of software products. That effort is reduced by allowing modeling artifacts to actually deliver more to the software product or set of software products under development than they do when used just for documentation purposes. Automated code generation from visual models is one of the main characteristics of MDD and the ultimate goal of the model transformation cycle. The other main characteristic of MDD is the reduction of models' sensitivity to change by (1) making them accessible and useful (therefore understandable in the first place) by all stakeholders; (2) changing models while the systems that rely on them are running; (3) storing the models in formats that other tools can use; and (4) automating the process of translating platform-independent models to platform-specific models and the former to code. Point 1 is achieved through notation, point 2 through dynamic language extension (through the runtime extension of the set of types available for modeling, which are the *language concepts* previously mentioned in this chapter), point 3 through interoperability and point 4 through user-definable mappings. An MDD infrastructure must provide for visual modeling and the means for defining visual modeling languages, which are abstract syntax, concrete syntax, well-formedness rules (constraints on the abstract syntax) and semantics. Such infrastructure must also provide for the use of OO (Object-Oriented) languages that allow extending the set of types available by those languages' APIs (Application Programming Interfaces) despite in a static way (not at runtime as MDD actually requires). Describing the previously mentioned concepts from the *language concepts metamodel* level, the concepts from the *language concepts* level and the also previously mentioned *user concepts* in a metalevel way (e.g. with the OMG modeling infrastructure) allows adding new *language concepts* dynamically at runtime. Finally an MDD infrastructure must provide for the means to define model transformations by the user in order to translate models ultimately into code of a specific implementation platform. A means to define model transformations is to use the model transformation languages QVT (Query/View/Transformation) (OMG, 2008) or ATL (ATLAS Transformation Language) (The Eclipse Foundation, 2010).

MDD relies on models that can be used as input to automated transformations (Swithinbank et al., 2005). In (Ruben & Vjeran, 2009) it is stated that the transformation of models into code can be facilitated by using software development patterns. The means to obtain that is to pack patterns as reusable assets with encapsulated implementation. We consider that a packed pattern can contain either the (pattern's) model and the code or just the model since not all patterns are to be directly converted into programming code. Depending on the type of pattern, it can be translated into code that can be directly included in the software solution under development in the programming environment for further manipulation or it can be imported in the modeling environment to be used in the modeling of the software solution by customizing the pattern's model elements and relating them with the remaining model elements. If the packed pattern contains the model and the code, then both the inclusion of the code in the software solution in the programming environment and the import of the model in the modeling environment can be performed. These ways patterns can be involved in the visual modeling of software systems and/or the automated code generation from visual models used in the development of those software systems just like MDD requires. According to (Greenfield & Short, 2004) a code template can be attached to the pattern to generate code from the model to which the pattern has been applied. Finally we consider that there is no point in using implementation patterns as packed patterns that can be imported in the programming environment as most of the times they depend on modeled elements parameters to be instantiated. In fact some of those patterns are already available in the programming environment through context menus of source code elements generated from models.

The models used to develop a software product or family of products evolve along the software development lifecycle and according to MDD end up in code. Pattern classifications help the actors involved in MDD software development processes to choose the most convenient patterns (in the form of models) to be incorporated into the models that are later transformed into code. By dividing patterns into categories all pattern classifications contribute to the use of patterns to develop software according to the MDD directives as the effort to select patterns without them would be higher, which would not contribute to the goal of MDD (raising the abstraction level at which software programs are written by reducing the software development effort). Patterns in the form of models also help raising the abstraction level at which software programs are written. Those that are not represented as models because they are to be only in code contribute to MDD by being considered in the process of automating code generation from visual models, during which the structure of code is thoroughly defined for the code that is generated from the visual models. For instance if the model from which we are to generate code incorporates the *Getter/Setter* pattern, we have to consider the implementation patterns like those in (Beck, 2008) applicable to the target platform in order to generate source code for the getters/setters (operations) (Swithinbank et al., 2005).

Especially the pattern classifications that reveal some kind of software development procedural notion contribute to MDD given that it is more likely that the most adequate patterns are selected. That is because those classifications avoid the wrong patterns to be handled by the wrong professionals, technologies and methodologies that make more sense in the context of a specific process' phase(s). Specific professionals, technologies and methodologies are more skilled to handle specific kinds of models that address specific kinds of problems in specific moments of MDD software development processes. This means that specific professionals, technologies and methodologies are more skilled to handle specific kinds of patterns (in the form of models) to be applied to the specific kinds of models they handle as input to the automatic generation of code. Those patterns address specific kinds of problems, which can be better understood by those professionals due to their skills and profile. The pattern classification we propose in this chapter is particularly based on a software development process, which is the RUP (Rational Unified Process) (Kruchten, 2000). The proposed pattern classification is also related to the OMG modeling infrastructure in the sense that it demands for the patterns to be classified according to the abstraction level at which they are represented (the OMG modeling infrastructure's levels M2, M1 or M0) for the reasons we will expose further on in this chapter.

THE MULTILEVEL AND MULTISTAGE CLASSIFICATION

Our multilevel and multistage pattern classification has three dimensions: the level (from the OMG modeling infrastructure), the Software Engineering discipline (based on the RUP) and the stage of the software development process (also based on the RUP). The classification adopts also an attribute, besides the three dimensions: the nature of the domain.

The Classification Explained

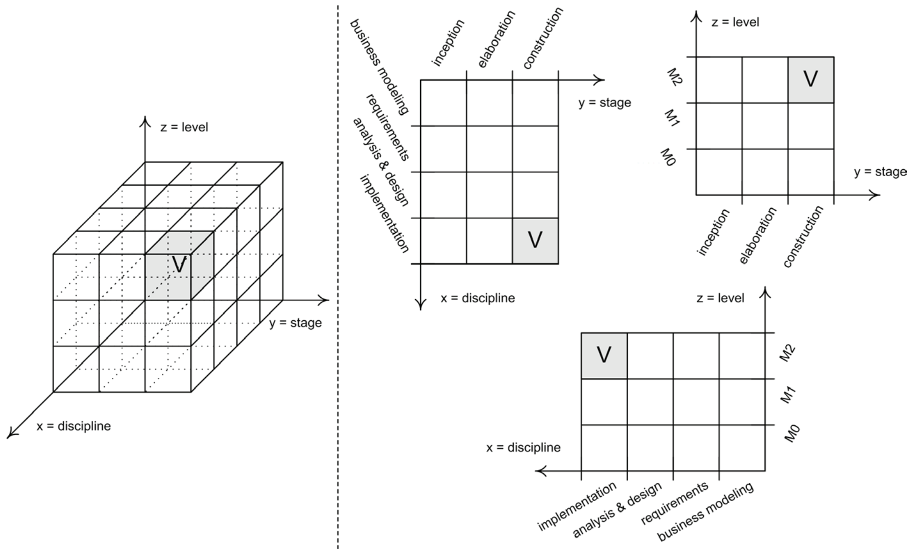
Domains can be of horizontal nature or of vertical nature. The vertical domains represent particular business domains and correspond to activity sectors (e.g. banking, insurance, trading, industry). The horizontal domains are traversal to the vertical domains, which means that they represent areas of knowledge common to every business domain (e.g. accounting, human resources, stock, project management). This does not mean that business applications (banking applications for example) shall contemplate all horizontal domains but it means that horizontal applications (for instance accounting applications) shall be usable by all the businesses possible, although there is a part of each horizontal domain that is only applicable to each business domain (e.g. there are accounting rules specific to the banking sector).

The multilevel character of our classification lies on the different levels of the OMG modeling infrastructure, which provides for a multilevel, four-layer modeling architecture. The classification's RUP-based Software Engineering discipline dimension provides for clear hints on the professionals who shall handle specific types of patterns, with particular technologies and methodologies. At last the classification's multistage character is given by the dimension associated with the RUP-based phases of the software development process. Our hypothesis is that the development of software can take more advantage of patterns and their proposed solutions if their adoption occurs at the right moment of the process of developing a software solution and within the context of the right Software Engineering professionals, technologies and methodologies, respecting the levels patterns shall follow throughout the adoption process, which involves dealing with models at different levels of abstraction as well. We consider that the positioning of patterns at the wrong category of any process-based classification leads to a misinterpretation of those same patterns, resulting in an unsuccessful adoption. By unsuccessful adoption we mean a constriction of the original patterns' advantages. Although our effort is towards minimizing the effects of pattern misinterpretation, pattern adaptation can still and will most likely occur over the pattern models we are going to expose in this chapter. Our classification (especially due to its multilevel character) reduces the chances of pattern misinterpretation since it reaches the metamodeling level (M2-level from the OMG modeling infrastructure). Unsuccessful pattern adoptions can lead to software solutions where the adopted patterns are unrecognizable.

Patterns vary in their abstraction level. Actually the same pattern may be positioned at different abstraction levels according to its representation. Normally the interpretation of a pattern is performed directly from the catalogue to the particular context of the product or the family of products. This way both the representation of the pattern in the catalogue and the interpretation of that same pattern are situated at the M1-level, which may not be adequate if the goal is to systematically use patterns and reduce the unsuccessful pattern adoptions during software production. Thinking about software families the matter with software product lines and software patterns may lie on the instantiation of M2 artifacts at the M1 layer, which again indicates the relevance of the abstraction level concerning the adoption of software patterns.

We are adopting the geometrical terminology to represent the pattern classification. Patterns can be positioned at the *pattern positioning geometrical space* placed in the first octant of the orthonormal referential like Figure 2 (on the left) shows. Actually that space may be partitioned into cubes. As patterns can be classified with three possible values according to two of the three axes of the referential and with four possible values according to the other axis, the pattern positioning geometrical space can be divided into 3×3×4 cubes as can also be seen from Figure 2 (on the left). The fourth criterion is the domain nature and in the case of the pattern positioned at the pattern positioning geometrical space in the figure it takes

Figure 2. Orthonormal referential with the dimensions of the multilevel and multistage classification on the axes plus the pattern categorization three-dimensional space (on the left). The projections of a pattern's positioning in a two-dimensional area (on the right)



the value vertical (V). That is why the grey cube representing the pattern is tagged with a *V* (the domain nature is not a dimension, it is an attribute so it does not correspond to an axis). Figure 2 (on the right) presents the projections of the pattern's positioning represented in a three-dimensional space on the left of the figure, this time in a two-dimensional area. The possible values of each dimension are attached to the axes. They will be detailed further on in this section of the chapter.

As we have already argued, leveraging patterns to the M2-level is a way of turning the decisions on the application of patterns more objective as well as of reducing the misinterpretation of patterns at the M1-level with all the disadvantages that subjective decisions and misinterpretation bring into the software development process and the quality of the software product itself. Multilevel instantiation shall occur during the adoption of patterns in order to systematize their use. Patterns are positioned at the pattern positioning geometrical space (according to the axes representing the Software Engineering disciplines and the OMG modeling infrastructure levels) with regards to their representations: the M2 model (pattern M2), the M1 model (pattern M1) and the M1 code. As we will see later on in this chapter, the pattern in the M1 representation is an instance of the pattern in the M2 representation, whereas the code is a transformation of the pattern's M1 model into a specific programming language code. The abstraction level decreases when moving from models at the M2-level to the code. Pattern catalogues represent patterns with M1 models and M1 code (source code). They do not propose patterns using their M2 representation (or metamodels). That is not our approach as it will be detailed in the next section of this chapter. The course of the artifacts inside the pattern positioning geometrical space as well as the course's projection on the discipline×level plan indicates that a small process within the whole software development process must occur when systematically dealing with patterns, which includes multilevel instantiation and transformation of models into code.

The reason for representing patterns in catalogues in their M1 representation is due to the willing of not compromising the applicability of patterns to a broader domain coverage. This is the risk of rising the abstraction level from M1 to M2. Naturally every risk has some potential for success and the risk of rising the abstraction level carries with it the advantage of turning the pattern adoptable by more domains. In order to adapt a pattern from a catalogue to a different domain than the one considered for representing the pattern in the catalogue it is necessary to know in which areas to change it and for that the pattern's structure has to be known as well. To know the structure of the pattern, the pattern has to be represented at the M2-level.

Although the pattern may assume several representations according to the level it is positioned at, we are talking about the same pattern since the diverse representations of the pattern answer to the same problem, within the same context, with the same solution, driven by the same recurrent and predictable forces (Beck, 2008; Gamma, Helm, Johnson, & Vlissides, 1995; Meszaros & Doble, 1997). Having various representations for the same pattern implies that the M2 representation of a pattern covers more functionality, therefore reaching higher levels of functional completeness than the M1 representation.

The Classification's Dimensions and Attribute

Next each criterion (the dimensions and the attribute) of the multilevel and multistage classification are described. As we have already stated the multilevel and multistage classification considers the moment of the software development process during which specific kinds of patterns, what we call pattern types (see section 3.3 for more information on the multilevel and multistage pattern types), shall be used. The *Discipline* dimension represents these different moments in the process of developing software. The multilevel and multistage classification considers as well the context in which patterns shall be used in terms of Software Engineering professionals, technologies and methodologies. Stages of software development are defined by different profiles of Software Engineering professionals who work with different kinds of technologies and methodologies tailored to their profiles. The *Stage* dimension represents these different stage-related professionals, technologies and methodologies in the process of developing software. Our classification takes also in a modeling infrastructure that has been adopted to avoid subjective decisions on the application of patterns, and situations of misinterpretation and corruption of patterns from catalogues while interpreting and adapting them respectively. The modeling infrastructure that has been considered is the OMG modeling infrastructure. The *Level* dimension represents the different levels of the OMG modeling infrastructure. Finally the multilevel and multistage classification takes into account that domain-specific artifacts for the development of families of software products are common these days, which means that the applicability of patterns to particular domain natures allows to choose between the patterns that are most adequate to a domain of a software product or family of products. The *Domain Nature* attribute represents the different (both) domain natures to which patterns are most applicable (or the applicability of patterns to both domain natures).

As the subtitles indicate, the *Discipline* dimension can take the values {*business modeling*, *requirements*, *analysis & design*, *implementation*} and the *Stage* dimension can take the values {*inception*, *elaboration*, *construction*}. The *Level* dimension corresponds to the levels of the OMG modeling infrastructure {M1, M2}. For now M3 is not being considered. We are not representing M3 in the figures because M3 can be represented with (UML) models and we haven't yet worked our classification at that level. Despite that, M0 is represented in the figures to remember the reader that after M1 code (compile-time code) we have M0 code (runtime code) but runtime code is not relevant to our classification. The *Domain Nature* attribute which has already been explained earlier in this section of the chapter can take the values {*vertical*, *horizontal*, *agnostic*}.

In order to use this classification do the following: (1) analyze the pattern you want to classify according to the dimensions *Discipline* and *Stage*, and give a value to each of those dimensions for that pattern you are classifying; (2) conclude on the pattern type (see section 3.3 for more information on the multilevel and multistage pattern types and how the dimensions *Discipline* and *Stage* determine the pattern type); (3) determine the pattern's level, which corresponds to giving a value to the *Level* dimension; (4) if the pattern is not represented in its M2 representation, draw an M2 model of the pattern; (5) by looking at the M2 representation of the pattern describe its semantics in textual form; and finally (6) by looking at the pattern's M2-level textual description and at the pattern's description in the catalogue classify the pattern in what its domain nature is concerned, which is equivalent to tagging the pattern with one of the three possible values for the *Domain Nature* attribute.

The assignment of patterns to particular chunks of the classification is dependent on the pattern type, therefore on the RUP's textual descriptions of its disciplines and phases (to conduct step 1). In order to determine the pattern's level the classifier (the subject who classifies) must be familiarized with the Four-Layer Architecture since he has to understand if the concepts the pattern presents are situated at the M2 or at the M1 levels. The classifier has to know the notion of multilevel instantiation. The classification process is dependent on the subject who conducts the process. Determining the pattern type is subjective as it implies looking at the textual descriptions of the RUP's disciplines and phases. Analyzing textual descriptions is subjective (at least in this approach). Determining the pattern's level is also subjective (at least in this approach) because it depends on the classifier's knowledge.

The Discipline Dimension

The RUP's Business Modeling Software Engineering Discipline

The RUP's *Business Modeling* discipline shall comprise activities of derivation of the software requirements the system to be developed must support in order to be adequate to the target organization and of analyzing how that system fits into the organization. The goal of the *Business Modeling* discipline is to model an organizational context for the system.

The RUP's Requirements Software Engineering Discipline

The RUP's *Requirements* discipline shall comprise activities of stakeholder request elicitation and of transformation of those requests into requirements on the system to be developed. Those requirements shall span the complete scope of the system. The requirements on what the system shall do have to be agreed with the stakeholders (customer and others). The goal of the *Requirements* discipline is to provide developers with a better understanding of the requirements the system must fulfill based on the customer's (or other stakeholder's) requests. It is also the goal of this discipline to delimit the boundaries of the system to be developed.

The RUP's Analysis & Design Software Engineering Discipline

The RUP's *Analysis & Design* discipline shall comprise activities of transformation of the requirements elicited with the stakeholders into a design of the system to be deployed. The design of the system shall contemplate an architecture for the system. The goal of this discipline is to specify the design of the system to be developed.

The RUP's Implementation Software Engineering Discipline

The RUP's *Implementation* discipline shall comprise activities of development, unit testing of the developed components and integration of the software components that will allow the system requested by the stakeholders to be deployed based on the design specifications elaborated in the context of the *Analysis & Design* discipline. When developing the system, the organization of the code shall be defined according to the layers of the subsystems to implement. Developing the system through components implies that all the components produced by different teams are integrated into an executable system. The goal of this discipline is to translate the design elements that came up in the context of the *Analysis & Design* discipline into implementation elements (source files, binaries, executable programs and others).

The Stage Dimension

The RUP's Inception Software Development Stage

The RUP's *Inception* stage shall comprise activities of discrimination of the critical use cases of the system and the primary operation scenarios vital to the design tradeoffs that will have to be made further on during the process. At least one candidate architecture shall be exhibited (and maybe demonstrated) and shall support the primary scenarios (or at least some of them) in order for the stakeholders to agree upon the fulfillment of the requests they exposed to the Software Engineers responsible for the requirements elicitation. The goal of this stage is to ensure that the software development project is both worth doing and possible to execute.

The RUP's Elaboration Software Development Stage

The RUP's *Elaboration* stage shall comprise activities of architecture handling like conceiving a baseline architecture of the system, thus providing a stable basis for the further design and implementation work which will take place during the *Construction* stage. This architecture shall contemplate and reflect the most significant requirements for the architecture of the system. Architectural prototypes shall be used to evaluate the stability of the architecture. The goal of this stage is to elaborate an architectural foundation for the upcoming detailed design and implementation efforts.

The RUP's Construction Software Development Stage

The RUP's *Construction* stage shall comprise activities of development of deployable software products from the baseline architecture of the system elaborated during the prior stage. The design, development and testing of all the requested functionality for the system shall be completed during this stage. The construction of the software system shall be conducted in an iterative and incremental way. It is during the construction of that software system that remaining use cases and other requirements are described, others are further detailed, the design built during the previous stage is enlivened and the implemented software is tested. The goal of this stage is to develop a complete software product ready to transition to the users.

The Level Dimension

The *Level* dimension of the classification corresponds to the levels of abstraction of the Four-Layer Architecture. Each model in the Four-Layer Architecture except for the one at the highest level is an instance of the one at the higher level. The M0-level refers to the data manipulated by software. The M1-level refers to models of user concepts. The M2-level refers to UML concepts models. These are models of models and so are called metamodels. A metamodel is a model whose elements are types in another model (an example of a metamodel is the UML metamodel). It describes the structure of the models, the elements that are part of those models and their respective properties. The meta-metamodels are at the highest level of the modeling infrastructure, the MOF (Meta-Object Facility) (OMG, 2006) or M3-level.

The Domain Nature Attribute

The *Domain Nature* attribute indicates whether the pattern is more adequate to vertical domains (industry, commerce, services and others) or to horizontal domains (accounting, stock, project management and others). Some patterns as it will be evidenced later in this chapter are domain nature agnostic, which means that they are applicable both to vertical and to horizontal domains.

THE PATTERN TYPES

Following are the pattern types from the multilevel and multistage classification. A pattern type represents a kind of pattern that has been classified with the same *Discipline* dimension's value and the same *Stage* dimension's value. A description is provided for each of the pattern types as well as the classification according to the *Discipline* and *Stage* dimensions. The classification of pattern types according to the *Level* dimension does not make sense as it depends on the representation of the pattern and has no influence on the definition of the pattern types themselves. The pattern types are: business patterns, analysis patterns, enterprise patterns, architectural patterns, design patterns and implementation patterns. These names have been chosen because they are the most common pattern names in the literature and make the most sense in our definitions of the pattern types.

This section will expose some examples of patterns that were classified with different pattern types. The patterns in this section suit the purpose of demonstrating how we have applied the multilevel and multistage classification of patterns. We provide for a representation of the patterns as M2-level (meta)models and as M1-level models (when applicable).

Be aware that some of the patterns we are going to analyze in this section have not been classified with the same pattern type name we have classified them with using our classification. For instance the *Posting* pattern has been classified as a business pattern by Pavel Hruby in (Hruby, 2006) but we classify it as an analysis pattern.

The Business Patterns

The term *business pattern* was inspired on IBM's definition of business pattern (Adams, Koushik, Vasudeva, & Galambos, 2001).

Business patterns are more pertinent in the context of vertical domains. They make the most sense to be handled during the *Inception* stage by professionals, technologies and methodologies from the *Business Modeling* and *Requirements* disciplines.

Business patterns are used to describe a solution to accomplishing a business objective. They shall address the users of the solution, the organization's software systems the users interact with (or the organization itself) and the organization's information (available through those systems or the organization itself). Business patterns may refer to e-business solutions that convey an organizational framing, validity and conformance of the solution to the business problem the solution is trying to solve. Software solutions shall be sustained by the business and this is achieved with the adoption of business patterns.

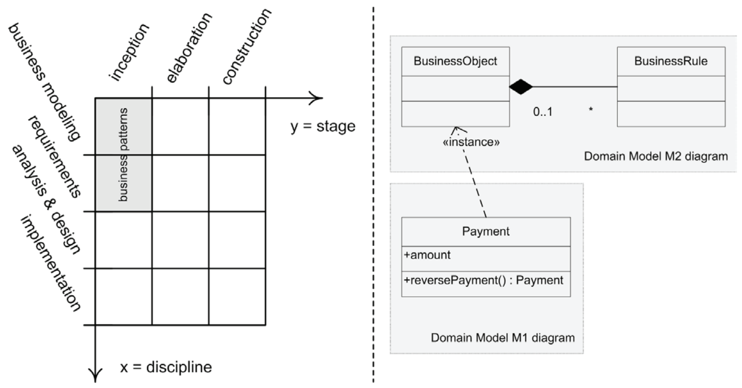
We can see examples of business patterns in (Adams, Koushik, Vasudeva, & Galambos, 2001) and also in (Eriksson & Penker, 2000).

Figure 3 (on the left) illustrates the positioning of business patterns according to the *Stage* and the *Discipline* dimensions.

The Domain Model Pattern

The *Domain Model* pattern's goal is to produce an object model of the domain or business area. A domain model must distinguish between the data the business involves and the business rules (or the rules used by the business). The behavior expressed by these business rules shall be placed in the business object that really needs it. Figure 3

Figure 3. The business patterns' positioning according to the Stage and the Discipline dimensions (on the left). The Domain Model pattern modeled at both the M2 and the M1 levels of the OMG modeling infrastructure (on the right)



(on the right) shows a model with an example of the *Domain Model* pattern in the M1 representation as well as the M2 representation of the pattern. The *Domain Model* pattern is composed of two types of concepts: business objects (or domain objects) and business rules. This is evidenced by the *Domain Model* M2 model in Figure 3 (on the right).

The *Domain Model* pattern suits the modeling of every business domain possible as every business domain has business objects and business rules on those objects. Even though the pattern is applicable to all business domains it is not appropriate to the modeling of a horizontal domain or to the modeling of structural business domain commonalities, which makes of it applicable to domains of vertical nature.

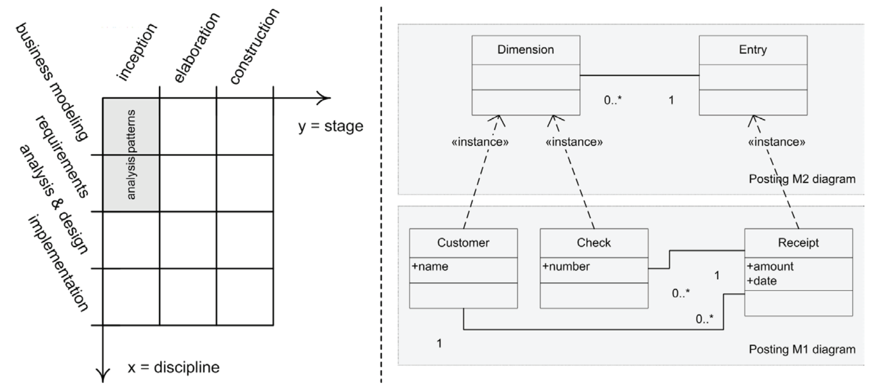
The *Domain Model* pattern does not show how to model objects or rules for a specific business domain but the types of concepts the pattern handles are business-related and shall be instantiated in order to model business domains. Besides and more important than that, the *Domain Model* pattern allows to model objects and rules that shall be handled by the solution to the business problem the solution is trying to solve. The *Domain Model* pattern is a very atomic pattern as it does not address the users of the solution or the organization's software systems the users interact with (or the organization itself); nonetheless it is adequate to reach the business domain model from the candidate architecture that shall be exhibited to the stakeholders. For all of these reasons we consider that the *Domain Model* pattern shall be classified as a business pattern.

By looking at the RUP's textual descriptions of its disciplines and phases, we concluded that the *Domain Model* pattern shall be used during the *Inception* software development stage and in the context of the *Business Modeling* and *Requirements* Software Engineering disciplines as seen in the previous section of this chapter. During the *Inception* stage a domain model must be built from a candidate architecture that translates the critical use cases and the primary operation scenarios. That domain model may be achieved with the application of the *Domain Model* pattern. The pattern shall help translating the requirements elicited with the stakeholders. Those requirements have to be adequate to the target organization, which is a concern of the *Requirements* discipline.

The Analysis Patterns

The term *analysis pattern* was inspired on Fowler's definition of analysis pattern (Fowler, 1997).

Figure 4. The analysis patterns' positioning according to the Stage and the Discipline dimensions (on the left). The Posting pattern modeled at both the M2 and the M1 levels of the OMG modeling infrastructure (on the right)



Analysis patterns are more applicable to horizontal domains. They shall be used during the *Inception* stage by professionals, technologies and methodologies from the *Business Modeling* and *Requirements* disciplines. In spite of being called analysis patterns it does not make sense to use them in the context of the *Analysis & Design* discipline. They have been called so because *analysis pattern* is a terminology spread out through the literature and also because Fowler's definition of analysis pattern inspired ours. In an older informal terminology the development of software was composed of three phases: analysis, design and implementation. With RUP formalizing the dimension of business modeling in the process of software development, analysis was divided into business modeling and requirements. The former design discipline corresponds to RUP's *Analysis & Design*.

Analysis patterns are solutions to recurrent problems in many (business) domains. They are composed of concepts that represent structural commonalities when modeling many different business domains.

We can see examples of analysis patterns in (Fowler, 1997).

Figure 4 (on the left) shows the positioning of analysis patterns according to the *Stage* and the *Discipline* dimensions.

Business patterns and analysis patterns are *dual patterns* since they coexist in the context of the *Inception* stage and of both the *Business Modeling* and the *Requirements* disciplines. Business patterns are not necessarily about software but they have to give input on how the software requirements of a business domain are adequate to an organization. Analysis patterns have to consider its adequacy to the target organization. They both have to be used during the earliest period of the software solution's development, when requirements are elicited and agreed with the stakeholders.

The Posting Pattern

Previously in (Hruby, 2006) the *Posting* pattern has been classified as a business pattern by Pavel Hruby. According to the multilevel and multistage classification the *Posting* pattern is classified as an analysis pattern. It is applicable to horizontal domains.

The point of the *Posting* pattern is to keep the history of economic events (commitments, contracts or claims) or in other words the history of interactions between economic agents for the exchange of economic resources like the purchase of products, the sale of services, invoices and corresponding payments, among others. Some examples of posting types are inventory posting, finance posting, man-hours posting and distance posting. Figure 4 (on the right) exposes a model with an example of the *Posting* pattern in the M1 representation as well as the M2 representation of the pattern. The *Posting* pattern contemplates two types of concepts: dimensions and entries. A posting dimension is either an economic agent or an economic resource. The purpose of the dimension is to provide additional information about the economic event or in other words provide descriptive information about the posting entries. A posting entry is an entry of a commitment, a contract or a claim. The purpose of the entry is to keep track of the history of economic events. In Figure 4 (on the right) we can see that Customer and Check are two posting dimensions of the posting entry Receipt. Most probably the Customer class represents the economic agent involved in the economic event represented by the entry class Receipt whereas the Check class represents the economic resource.

The *Posting* pattern is constituted by concepts belonging to a horizontal domain (the accounting domain). Nevertheless the *Posting* pattern has only the concept of posting entry in common with the *Accounting* pattern (in the *Accounting* pattern the concept of posting entry corresponds to the concept of agreement).

The arguments for classifying the *Posting* pattern as an analysis pattern as well as for its adequacy to the *Inception* software development stage and the *Business Modeling* and *Requirements* Software Engineering disciplines are the same we described beforehand for the *Accounting* pattern.

The Enterprise Patterns

The term *enterprise pattern* was inspired on Fowler's considerations about enterprise patterns and enterprise software in (Fowler, 2009).

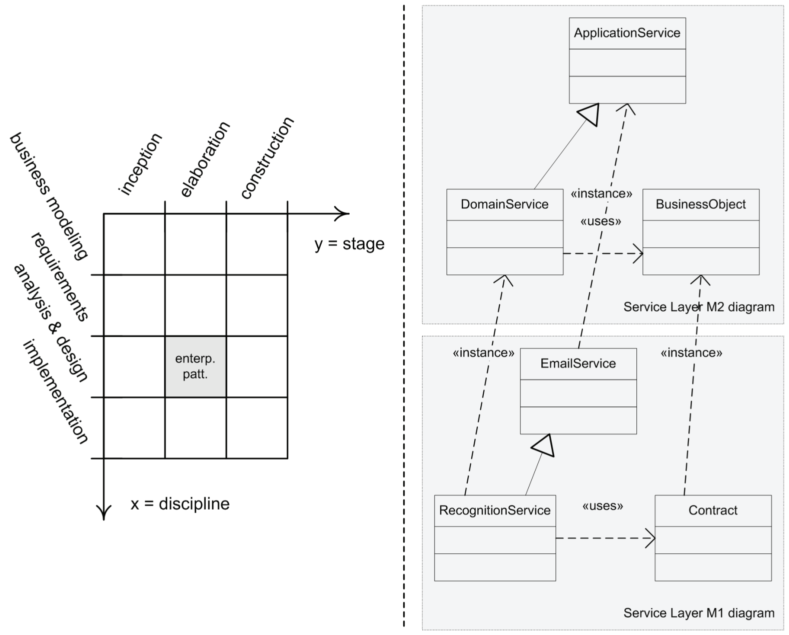
Enterprise patterns are most adequate to vertical domains. They are more relevant in the context of the *Elaboration* stage by professionals, technologies and methodologies from the *Analysis & Design*discipline.

Enterprise patterns are used in the development of software systems on which various businesses rely on and run (the so called enterprise software systems). Normally the architecture of such systems is a layered architecture. Conception decisions on layered architectures are design decisions that have to be taken inside a logical layer or between different logical layers. Often single enterprise applications need to interact so enterprise patterns have also to propose solutions to the integration of enterprise applications problem. Validations, calculations and business rules on the data an information system manipulates vary according to the domain and change as the business conditions change. Enterprise applications must respond to ever changing business requirements.

Enterprise patterns address architectural concerns as well as the architecture patterns we will be talking next but whereas enterprise patterns are mainly concerned with topological architecture, architectural patterns are mainly concerned with logical architecture.

This chapter does not consider the notion of *enterprise* as the RUP does not consider it. The RUP is a Software Engineering process framework. IBM has delivered a RUP plug-in called RUP SE (RUP for Systems Engineering) (Cantor, 2003). The RUP SE has enlarged the RUP with the consideration that the development of large-scale systems must be concerned with software, hardware, workers and information. The RUP SE considers different perspectives on

Figure 5. The enterprise patterns' positioning according to the Stage and the Discipline dimensions (on the left). The Service Layer pattern modeled at both the M2 and the M1 levels of the OMG modeling infrastructure (on the right)



the system (logical, physical, informational, and others). The RUP SE is shortly a framework for addressing the overall system's issues. The RUP SE addresses behavioral requirements (the way the system shall behave in order to fulfill its role in the enterprise). The RUP does not express such concern with the enterprise in which the system will play its role. In fact this kind of concern is more from the field of Systems Engineering than from the field of Software Engineering. When we talk about system requirements in the context of Software Engineering we are specifically talking about software system requirements. The system requirements are derived from an understanding of the enterprise, its services and the role that the system (software-based or not) plays in the enterprise. For instance the RUP SE suggests that the enterprise shall be partitioned into the system and its actors in order to derive the system requirements. In the RUP SE an enterprise is faced as a set of collaborating systems that collaborate to realize enterprise services, mission and others. The system attributes are obtained from an analysis of the enterprise needs. As this chapter talks about software system development patterns in the context of RUP (not RUP SE), this chapter is related to Software Engineering, not to Systems Engineering, which means that this chapter's *enterprise patterns* have nothing to do with the concept of *enterprise* from the Systems Engineering field. The term *enterprise pattern* comes from the term *enterprise application architectural pattern* from Folwer's book "Patterns of Enterprise Application Architecture" (Fowler, 2003b).

We can see examples of enterprise patterns in (Fowler, 2003b).

Figure 5 (on the left) depicts the positioning of enterprise patterns according to the *Stage* and the *Discipline* dimensions.

The Service Layer Pattern

In (Fowler, 2003b) Fowler classified the *Service Layer* pattern as an enterprise application architectural pattern. According to the multilevel and multistage classification the *Service Layer* pattern is classified as an enterprise pattern.

The purpose of the *Service Layer* pattern is to provide for operations to access the enterprise application's stored data and business logic. The *Service Layer* pattern can be implemented with a set of facades over a domain model. The classes implementing the facades do not implement any business logic, which is implemented by the business object's rules from the domain model. The facades gather the operations the application has available for interaction with client layers. The Service Layer can also be implemented with classes directly implementing the application logic and delegating on business object classes for domain logic processing. Application logic is grouped into classes of related application logic. These classes are application service classes. Figure 5 (on the right) depicts an example of this second strategy for implementing the *Service Layer* pattern at the modeling level. The figure shows a model with an example of the *Service Layer* pattern in the M1 representation as well as the M2 representation of the pattern. As we may conclude from the figure the *Service Layer* pattern is composed of two types of concepts: application services and domain services. Business objects are also represented in the models as the domain services rely on them for business logic. The domain services act as intermediates between the application services and the business objects since they provide for calls to application logic in application services and for calls to business logic residing on business objects. These last calls are made inside the service operations the domain services provide for, which correspond to the use cases the actors want to perform with the application.

As the main focus of the *Service Layer* is the domain service acting as a bridge between the application logic and the business logic, and not implementing any business domain logic (just accessing it) we have tagged this particular enterprise pattern as domain nature agnostic.

The *Service Layer* pattern has been classified in this chapter as an enterprise pattern because it is used to develop enterprise software systems for specific business domains. When developing enterprise applications, logical layers are essential and the concern of the *Service Layer* pattern (to separate application logic from business logic) proves that we talking about an enterprise pattern.

By looking at the RUP's textual descriptions of its disciplines and phases we concluded that the *Service Layer* pattern shall be used during the *Elaboration* software development stage and in the context of the *Analysis & Design* Software Engineering discipline. Since splitting application logic from business logic is an architectural decision with impacts at the level of the baseline software system architecture it makes sense to adopt the *Service Layer* pattern during the *Elaboration* stage and by the professionals, technologies and methodologies responsible for the software design.

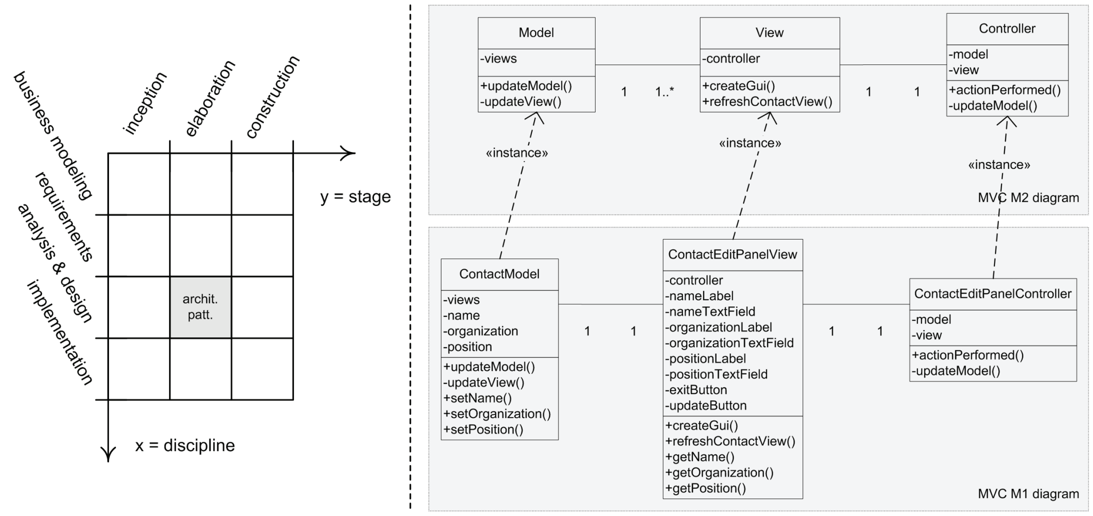
The Architectural Patterns

The term *architectural pattern* was inspired on Buschmann, *et al.* and Zdun (Buschmann, Meunier, Rohnert, Sornmerlad, & Stal, 1996; Zdun & Avgeriou, 2005).

Architectural patterns are more appropriate to horizontal domains. They shall be picked up from catalogues for usage during the *Elaboration* stage by professionals, technologies and methodologies from the *Analysis & Design* discipline.

Architectural patterns are used in the definition of the structure of software solutions. The architecture of a system is the design artifact that represents the functionality-based structure of that system and shall address quality or non-functional attributes wished-for the system. Architectural patterns shall help improving both the functional and the quality attributes of software systems.

Figure 6. The architectural patterns' positioning according to the Stage and the Discipline dimensions (on the left). The MVC pattern modeled at both the M2 and the M1 levels of the OMG modeling infrastructure (on the right)



We can see examples of architectural patterns in (Buschmann, Meunier, Rohnert, Sornmerlad, & Stal, 1996).

Figure 6 (on the left) shows the positioning of architectural patterns according to the *Stage* and the *Discipline* dimensions.

Enterprise patterns and architectural patterns are *dual patterns* since they coexist in the context of the *Elaboration* stage and of the *Analysis & Design* discipline.

The Model-View-Controller Pattern

Originally in (Buschmann, Meunier, Rohnert, Sornmerlad, & Stal, 1996) the MVC (*Model-View-Controller*) pattern has been classified by Buschmann, *et al.* as an architectural pattern. According to the multilevel and multistage classification the MVC pattern is classified as an architectural pattern. It is adequate to both horizontal and vertical domains, so it is agnostic relatively to the domain nature.

The purpose of the MVC pattern is to ensure the consistency between the user interface and the business information of a software system. The separation of the user interface from the business information of a software system provides for user interface flexibility. Figure 6 (on the right) depicts an example of a model of the MVC pattern in the M1-level and also the MVC pattern represented in the M2-level. The MVC pattern is composed of three types of classes: a model, a view and a controller. The model contains the business information that is to be presented to the user. The view obtains the information from the model and displays it to the user. The controller is responsible for requesting the business information updating on the model upon user action (event) on the graphical interface (view). It takes the business information from the view and requests for the model's updating with that information.

Although the model component contains business information the MVC pattern is adequate to both horizontal and vertical domains, which makes of it agnostic in what its domain nature is concerned. The pattern can either be adopted if the business information is relative to horizontal business objects or to vertical business objects.

The MVC pattern is classified as an architectural pattern according to the multilevel and multistage classification since it is used to define the structure of the software system, namely the structure of the client-side of the system. The pattern allows for the software system to be flexible concerning its user interface, which is a quality attribute wished-for that system. Mainly the MVC pattern is responsible for the structure of the client-side of the software system in order for it to be able to update business information upon events triggered by the user on the user interface (which allows the system to provide for the update functionality to the user).

By looking at the RUP's textual descriptions of its disciplines and phases we concluded that the MVC pattern shall be used during the *Elaboration* software development stage and in the context of the *Analysis & Design* Software Engineering discipline. As the MVC pattern is used to define the structure of the client-side of the system, addressing both the update functionality and the user interface flexibility (non-functional requirement), it shall be part of the system's architecture, which shall be part of the system's design specification. The system's baseline architecture shall contemplate the most significant architectural requirements, and the MVC pattern addresses the consistency between the user interface and the business information of the software system (which is a requirement vital to interactive software systems).

The Design Patterns

The term *design pattern* was inspired on the GoF's patterns (Gamma, Helm, Johnson, & Vlissides, 1995).

Design patterns are domain nature agnostic, which means that they are both applicable to vertical and to horizontal domains. They shall be manipulated during the *Construction* stage by professionals, technologies and methodologies from the *Analysis & Design* discipline.

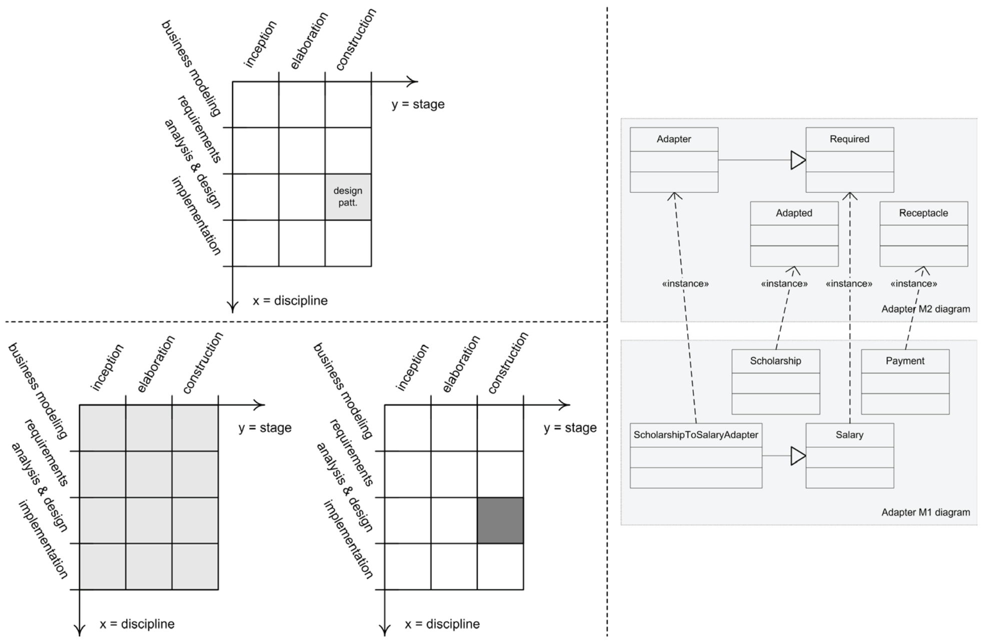
Although the GoF described design patterns as OO software patterns we consider design patterns are those that are applicable to the refinement or detailing of the software system architecture. For instance Larman's GRAS (General Responsibility Assignment Software) (Larman, 2001) patterns are design patterns since they have to do with behavioral aspects that only come up during a mechanistic design phase of the software solution's development (by mechanistic we mean structural or behavioral mechanisms more refined than components from logical architectures) (Larman, 2001).

The presence of code in design patterns is only to give examples. Design patterns are independent of the languag, as we can see from the GoF catalogue (Gamma, Helm, Johnson, & Vlissides, 1995) (they only talk about OO concepts, not language features). The sample code section provides for code to illustrate the example given in the motivation section, where the reader is given a scenario to illustrate a design problem in order for him to better understand the more abstract description of the pattern that follows the motivation section. Again the code is an illustration of the pattern's applicability.

Figure 7 (on the top left) depicts the positioning of design patterns according to the *Stage* and the *Discipline* dimensions.

Figure 7 (on the bottom left) illustrates the difference between our definition of design pattern and GoF's. The lighter grey area corresponds to the pattern positioning space of the GoF catalogue. The darker grey area corresponds to the pattern categorization area of our classification where we position *our* design patterns. These areas have been drawn taking only the *Discipline* and the *Stage* dimensions into consideration as the *Level* dimension does not allow demonstrating the difference between both definitions. We consider that design patterns shall only be used during the *Construction* stage of the software development process as the Software Engineering professionals, technologies and methodologies of the *Analysis & Design* are the most adequate to handle these patterns due to their professional

Figure 7. The design patterns' positioning according to the Stage and the Discipline dimensions (on the top left). The difference between our design patterns and the GoF's according to the Stage and the Discipline dimensions (on the bottom left). The Adapter pattern modeled at both the M2 and the M1 levels of the OMG modeling infrastructure (on the right)



profile and adequacy to the *Construction* stage's activities and goals. We also consider that if design patterns are handled throughout the whole software development stages and by the people and tools (technologies and methodologies) of every Software Engineering discipline, the advantages predicted in pattern catalogues of the adopted design patterns are not going to be preserved and that the design patterns in the catalogues are not going to be used in their full potential by the people most skilled to handle them.

The Adapter Pattern

In the past in (Gamma, Helm, Johnson, & Vlissides, 1995) the *Adapter* pattern has been classified as a design pattern by the GoF but in the sense of OO software pattern. According to the multilevel and multistage classification the *Adapter* pattern is classified as a design pattern. It is applicable to both horizontal and vertical domains, which makes of it a domain nature agnostic pattern.

The *Adapter* pattern (also known as *Wrapper*) has to do with a class converting the interface of one class to be what another class expects. Figure 7 (on the right) shows a model exemplifying the *Adapter* pattern in its M1 representation as well as the M2 representation of the pattern. This is what the *Adapter*'s implementation described at the M2-level should look like: "The Adapter must have an input parameter of the Adapted's type in its constructor and extend the Required and call the Adapted's appropriate operation inside the operation required by the Receptacle". The *Adapter*'s description at the M2-level in terms of semantics is the following: "The Receptacle requires the Adapted to be adapted to the Required through the Adapter (the process is called *adaptation*). The goal is for the Receptacle to be able to call the Required's operation from an instance of the Adapted".

The *Adapter* pattern is independent from any domain (or domain nature agnostic) because the adapter, the adapted, the required and the receptacle objects can belong to every domain possible. As long as the semantics or business logic (at the M1-level) specific of a certain domain complies to the M2 semantics we described in the previous paragraph the *Adapter* pattern is applicable to that domain no matter what the business is.

The *Adapter* pattern deals with classes and their operations that implement the interface operations those classes are expected to implement. Essentially the contents of those operations that are of relevance to the *Adapter* pattern are calls to other operations. As we can see we are not arguing about business logic implemented by the class' operations but rather about the structure of the classes targeted by the adaptation, which means we are discussing structural aspects rather than behavioral. Nevertheless and once again the *Adapter* pattern shall be applied during the mechanistic design phase of the system's development when classes shall be derived from architectural components. The *Adapter* pattern in its semantics shall be used to detail the baseline software system architecture and be part of a design specification containing the interface design of the classes involved in the *adaptation* process. For all of these reasons we have classified the *Adapter* pattern as a design pattern.

By looking at the RUP's textual descriptions of its disciplines and phases we concluded that the *Adapter* pattern shall be used during the *Construction* software development stage and in the context of the *Analysis & Design* Software Engineering discipline as already argued in this chapter. The adequacy of such stage and discipline to the *Adapter* pattern is intimately related to the reasons we have just exposed for classifying the *Adapter* pattern as a design pattern.

The Implementation Patterns

The term *implementation pattern* was inspired on Beck's definition of implementation pattern (Beck, 2008).

Implementation patterns are domain nature agnostic. They shall be considered during the *Construction* stage by professionals, technologies and methodologies from the *Implementation* discipline.

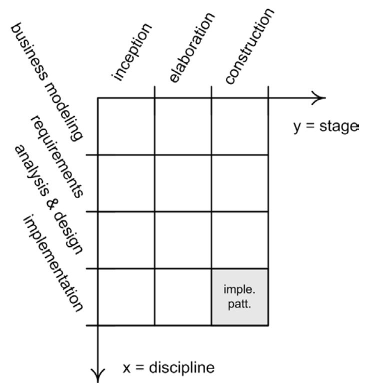
Implementation patterns are in fact the patterns in Kent Beck's catalogue (Beck, 2008) for instance and not Java or other language-specific patterns. The difference between design patterns and implementation patterns is that as Kent Beck claimed (Beck, 2008) design patterns are applicable a few times in the day of a programmer whereas his implementation patterns are applicable every few seconds in the day of a programmer. He also claimed that his implementation patterns teach readers how to use certain OO language constructs regardless of the language (despite him using a trivial subset of Java to exemplify the patterns). Java patterns or other language-specific patterns are just a different representation of design patterns (Grand, 2002; Stelting, 2002) (e.g. in (Stelting, 2002) Java is *applied* to the GoF patterns and other patterns). A different representation changes the pattern's level in the classification (e.g. in the case of the patterns from (Stelting, 2002) they had to be situated at the M1 (code) level in order for them to be called *Java* patterns). Kent Beck refers his patterns are applicable when all domain-specific questions are solved and developers are left with solely technical issues.

Figure 8 illustrates the positioning of implementation patterns according to the stage and the discipline dimensions.

The Value Object Pattern

In (Beck, 2008) the *Value Object* pattern has been classified as a class pattern. In the context of the multilevel and multistage classification the *Value*

Figure 8. The implementation patterns' positioning according to the stage and the discipline dimensions



Object pattern is classified as an implementation pattern. It is adequate to both horizontal and vertical domains, which means that it is domain nature agnostic.

The purpose of the *Value Object* pattern is to create objects that once created cannot have the values of the variables they handle changed. The solution is to set the value of those variables when the object is created through its constructor. No other assignments shall be made to those variables elsewhere in the object's class. Operations on the object shall always return new objects that shall be stored by the requester of the operation. Shortly value objects are objects representing mathematical values, which are values that do not change over time (have no state). For instance a transaction (value object) shall not change over time, rather an account changes over time (a transaction implies a change of state in the account). It does not make sense to model implementation patterns as they are only to exist in code, not in models, which implies that they are always represented at the M1-level (compile-time code).

The *Value Object* pattern shall be involved in the coding of both horizontal and vertical domain software systems since it is about the construction of objects that shall not change over time, the assignment of values to those objects' variables and the operations on those (value) objects.

The *Value Object* pattern has been classified as an implementation pattern because it is about the technical details of using classes (an OO language construct), in this case to create objects that shall have no state (whose variables' values shall not change over time).

By looking at the RUP's textual descriptions of its disciplines and phases we concluded that the *Value Object* pattern shall be used during the *Implementation* software development stage and in the context of the *Construction* Software Engineering discipline as previously mentioned in this chapter. The *Value Object* is related to the development of software systems, particularly to the development of implementation elements (source code).

FUTURE RESEARCH DIRECTIONS

Future work concerning the software patterns in the context of the software development process involves studying how patterns evolve over the time of that process. This evolution demands for the comprehension of the relationships between software patterns (especially those positioned at consecutive stages). It also demands for the analysis of how time implies that software patterns are associated with each other in a chain. The gap between patterns used at different stages shall be bridged in order to have a complete multistage software development process that contemplates different artifacts (software patterns and other artifacts like use case models, component models and others). In fact software patterns used at different stages solve the same problem at different levels of abstraction.

Software patterns may be used to detail logical software system architectures (expressed through component models). As software patterns are normally presented in class models, the detailing of those architectures requires knowing how to apply the concept of class to the concept of (logical) component.

The consideration of software patterns within the context of the software development process claims for the specialization of the actors who intervene in that process with specific roles during the adoption of those patterns. It is relevant to study the impacts of other software development processes (besides the RUP) in the proposed pattern classification.

Developing software product lines with software patterns (and other artifacts) may have some particular implications. Some variability mechanisms may have to be taken into account in software patterns. The use of those mechanisms may be constrained to a specific level of the OMG modeling infrastructure (the M2-level) and to specific pattern types. It may be necessary to define all the possible M2-level concepts (e.g. classes, attributes, operations) and/or the values of those concepts (e.g. class names, class attributes, class operations) as well as the application of all of them to all or some of the product line's members. The whole matter with software product lines and software patterns may mainly lie on the instantiation of M2-level artifacts at the M1-level.

Finally it is important to determine which software patterns may and shall be made available in modeling infrastructures (either through libraries of software pattern metamodels or models, or through domain-specific languages).

CONCLUSION

Some lessons have been learned on the application of the multilevel and multistage pattern classification to some patterns from the literature. After looking at the RUP's textual descriptions of its disciplines and phases some patterns were not classified with the pattern type we expected they would be classified with. This means that a procedural referential such as the RUP is important to classify patterns, mainly because it gives the classification a notion of software development process. It also means that the awareness of the adequacy of a pattern in a catalogue to a specific discipline and stage changed after the multilevel and multistage pattern classification has been elaborated. Initially before an in depth analysis of the RUP's textual descriptions and the definitions of the various pattern types we expected that (1) analysis patterns did not make sense in the context of the RUP's *Business Modeling* discipline; (2) design patterns made sense in the context of both the RUP's disciplines of *Analysis & Design* and *Implementation*, and of both the RUP's *Elaboration* and *Construction* stages; and (3) patterns that could be contextualized in the RUP's *Implementation* discipline and in the RUP's *Construction* phase were language-specific patterns. After analyzing RUP's textual descriptions and the pattern type definitions we concluded that (1) analysis patterns do make sense in the context of the RUP's *Business Modeling* discipline; (2) design patterns make only sense in the context of the RUP's *Analysis & Design* discipline and the RUP's *Construction* stage; and (3) language-specific patterns are a translation of design patterns into some language, not implementation patterns.

One of the reasons that was in the genesis of the creation of the multilevel and multistage classification was to provide for some guidance on the adoption of software development patterns in order to avoid loosing the original advantages of the pattern throughout the adoption process. For this reason we have considered that the pattern classification had to rely on the software development process. The benefits of such an approach to pattern classification are: (1) the knowledge of the moment from the software development process in which to use specific kinds of patterns; and (2) the knowledge of who the Software Engineering professionals most skilled to handle those specific kinds of patterns in each stage of the software development process are, considering their instruments (technologies and methodologies).

The systematic character of the multilevel and multistage classification is based on the objectiveness of the decisions on the application of software development patterns, which may be assured with the adoption of a modeling infrastructure. A systematic use of software development patterns is likely to also prevent the misinterpretation and corruption of patterns from catalogues when interpreting and adapting them respectively.

Besides being concerned with the stages and the Software Engineering professional's skills and the instruments they handle to conduct Software Engineering activities, and besides translating concerns with the systematic use of software development patterns the multilevel and multistage classification is also concerned with the nature of the domain, which is one of the criteria that compose the classification. Therefore the multilevel and multistage classification is focused on domain-based software development. The classification also focuses on model-driven software development since it incorporates through its multilevel character the OMG modeling infrastructure by considering that patterns can be represented at different levels of that infrastructure, which influences their interpretation.

The multilevel and multistage pattern classification is innovative in some ways relatively to the existing literature. Most pattern classifications do not classify patterns based on the software development process. The only classification that does, disregarded the analysis phases (business modeling and requirements) of the software development process. The multilevel and multistage classification though addresses business modeling and requirements.

REFERENCES

Adams, J., Koushik, S., Vasudeva, G., & Galambos, G. (2001). *Patterns for e-Business: A Strategy for Reuse*. Indianapolis, Indiana: IBM Press.

Atkinson, C., & Kühne, T. (2003). Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, *20*(5), 36–41. doi:10.1109/MS.2003.1231149

Beck, K. (2008). *Implementation Patterns*. Upper Saddle River, NJ: Addison-Wesley.

Buschmann, F., Henney, K., & Schmidt, D. C. (2007a). *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Hoboken, NJ: Wiley.

Buschmann, F., Henney, K., & Schmidt, D. C. (2007b). *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. Hoboken, NJ: Wiley.

Buschmann, F., Meunier, R., Rohnert, H., Sornmerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. Hoboken, NJ: Wiley.

Cantor, M. (2003). Rational Unified Process for Systems Engineering - Part III: Requirements Analysis and Design. *The Rational Edge*, from http://www.ibm.com/developerworks/rational/rationaledge

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., & Little, R. (2002). *Documenting Software Architectures: Views and Beyond*. Upper Saddle River, NJ: Addison-Wesley.

Eriksson, H.-E., & Penker, M. (2000). *Business Modeling With UML: Business Patterns at Work*. Hoboken, NJ: Wiley.

Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Upper Saddle River, NJ: Addison-Wesley.

Fowler, M. (2003a). Patterns. *IEEE Software*, *20*(2), 56–57. doi:10.1109/MS.2003.1184168

Fowler, M. (2003b). *Patterns of Enterprise Application Architecture*. Upper Saddle River, NJ: Addison-Wesley.

Fowler, M. (2009). *Patterns in Enterprise Software* from http://martinfowler.com/articles/enterprisePatterns.html

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Upper Saddle River, NJ: Addison-Wesley.

Grand, M. (2002). *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*. Hoboken, NJ: Wiley.

Greenfield, J., & Short, K. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Hoboken, NJ: Wiley.

Hruby, P. (2006). *Model-Driven Design Using Business Patterns*. Berlin: Springer-Verlag.

Kircher, M., & Jain, P. (2004). *Pattern-Oriented Software Architecture: Patterns for Resource Management*. Hoboken, NJ: Wiley.

Kruchten, P. (2000). *The Rational Unified Process: An Introduction*. Upper Saddle River, NJ: Addison-Wesley.

Larman, C. (2001). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Upper Saddle River, NJ: Prentice Hall.

Meszaros, G., & Doble, J. (1997). A Pattern Language for Pattern Writing. In Martin, R. C., Riehle, D., & Buschmann, F. (Eds.), *Pattern Languages of Program Design 3* (pp. 529–574). Upper Saddle River, NJ: Addison-Wesley.

OMG. (2006). *Meta-Object Facility: Core Specification - version 2.0* from http://www.omg.org

OMG. (2008). *Meta Object Facility 2.0 Query/View/Transformation: Specification - version 1.0* from http://www.omg.org

OMG. (2009a). *Object Management Group* from http://www.omg.org

OMG. (2009b). *Unified Modeling Language: Superstructure - version 2.2* from http://www.omg.org

Pree, W. (1995). *Design Patterns for Object-Oriented Software Development*. Upper Saddle River, NJ: Addison-Wesley.

Ruben, P., & Vjeran, S. (2009). Framework for Using Patterns in Model-Driven Development. In Papadopoulos, G. A., Wojtkowski, G., Wojtkowski, W., Wrycza, S., & Zupančič, J. (Eds.), *Information Systems Development: Towards a Service Provision Society* (pp. 309–317). Berlin, Heidelberg: Springer-Verlag.

Schmidt, D., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Hoboken, NJ: Wiley.

Soukup, J. (1995). Implementing Patterns. In Coplien, J. O., & Schmidt, D. C. (Eds.), *Pattern Languages of Program Design* (pp. 395–412). Upper Saddle River, NJ: Addison-Wesley.

Stelting, S. (2002). *Applied Java Patterns*. Upper Saddle River, NJ: Prentice Hall.

Swithinbank, P., Chessell, M., Gardner, T., Griffin, C., Man, J., & Wylie, H. (2005). *Patterns: Model-Driven Development Using IBM Rational Software Architect*. Indianapolis, Indiana: IBM Press.

The Eclipse Foundation. (2010). ATL Project from http://www.eclipse.org/m2m/atl

Tichy, W. F. (1997). *A Catalogue of General-Purpose Software Design Patterns*. Paper presented at the 23rd Technology of Object-Oriented Languages and Systems (TOOLS-23), Santa Barbara, California, USA.

Zdun, U., & Avgeriou, P. (2005). *Modeling Architectural Patterns Using Architectural Primitives*. Paper presented at the 20th Annual ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), San Diego, California, USA.

Zimmer, W. (1995). Relationships between Design Patterns. In Coplien, J. O., & Schmidt, D. C. (Eds.), *Pattern Languages of Program Design* (pp. 345–364). Upper Saddle River, NJ: Addison-Wesley.

KEY TERMS AND DEFINITIONS

Software pattern: reusable solution to a problem that occurs often throughout the software development process.

Pattern Classification: Activity of organizing patterns into groups of patterns that share a common set of characteristics.

Multilevel Software Development Process: Software development process concerned with the levels of abstraction in which the different artifacts involved in the development of software are handled.

Multistage Software Development Process: Software development process composed of some stages organized in a consecutive temporal order.

Pattern Adoption: Set of activities that consist of using the pattern somehow when producing software artifacts.

Pattern Interpretation: Activity that consists of reading the pattern from the pattern catalogue and reasoning about the solution the pattern is proposing for that problem in that given context.

Pattern Adaptation: Activity of modifying the pattern from the catalogue without corrupting it (corrupting the pattern includes corrupting the pattern's semantics and the pattern's abstract syntax).

Pattern Application: Actual use of a pattern in the development of software, whether to develop software applications or families of software applications, or to inspire the conception of design artifacts.

Multilevel Instantiation: Instantiation of M2-level patterns at the M1-level during the adoption of patterns.

Multilevel and Multistage Pattern Classification: Pattern classification concerned with the levels of abstraction in which the different software patterns are handled and composed of some stages organized in a consecutive temporal order.