ON VERIFYING THE USE OF A PATTERN LANGUAGE IN MODEL DRIVEN DESIGN

Bahman Zamani

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy (Computer Science)
Concordia University
Montréal, Québec, Canada

July 2009

Your file Votre référence
ISBN: 978-0-494-63401-1
Our file Notre référence
ISBN: 978-0-494-63401-1

Abstract

On Verifying the Use of a Pattern Language
in Model Driven Design

Bahman Zamani, Ph.D.
Concordia University, 2009

This thesis addresses the problem of verifying the application of a Pattern Language in a design that is built based upon the patterns of the language in a Model-Driven approach. Exploiting the ideas of compilers, we propose a process named Pattern Language Verifier (PLV). We argue that building a PLV for a given Pattern Language, requires the Structural, Syntactic, and Semantic rules of the language to be precisely defined. We present three formalisms for defining these three groups of rules. PLV is a profile-driven process and assumes that a UML Profile is already defined for the underlying Pattern Language.

PLV consists of four phases: Pattern Structural Verifier (PSV), Pattern Language Syntactic Verifier (PTV), Pattern Language Semantic Verifier (PMV), and Pattern Language Advisor (PLA). PSV verifies the structure of every single pattern used in the design model. PTV verifies the relationships between the detected patterns. PMV verifies the semantic aspects of the patterns. PLA reports the problems to the designer and guides him/her in fixing the errors.

For the case study, a group of enterprise architectural patterns is selected as the Pattern Language. The Structural, Syntactic, and Semantic rules of the language are defined using the proposed formalism, and a UML Profile is defined for the language. A PLV is designed and implemented as an integration into an open source modeling tool. The tool is then utilized in designing a sample web application: Online Student Registration System. The usefulness of the tool is represented by walkthrough scenarios that show finding the mistakes in the model and helping the designer repair the detected problems.

Acknowledgments

A thesis like this would never be accomplished without support and encouragement of others.

I am extremely indebted to my supervisor, Dr. Greg Butler, for his teaching, supervision, and patience during last five and a half years of my study. I learned a lot from him.

I would like to thank the members of my committee from Concordia University, Dr. Peter Grogono, Dr. Patrice Chalin, Dr. Juergen Rilling, and Dr. Ferhat Khendek for their fruitful feedback during my doctoral proposal and my defense.

I am thankful to Dr. Peter Hitchcock, from Dalhousie University, for his time to serve as external examiner of my defense, and for his valuable recommendations on the thesis.

I wish to thank the administrative staff of our Department, particularly, the Graduate Programs Advisor, Halina Monkiewicz, the Laboratory Coordinator, Pauline Dubois, and the Diploma & Certificate Program Assistant, Edwina Bowen.

I wish to thank all my friends and colleagues, especially, Farzad Kohantorabi and Stuart Thiel for the discussion about the patterns and their valuable suggestions. I also thank Sahar Kayhani who contributed to parts of the implementations.

I would express my warmest gratitude to my parents for their prayers.

And last but not least, I would like to thank and express my gratitude to my beloved family, my wife Farzaneh Kazemi, my daughter Sahar, and my son Sajjad, for all their support and love.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
CASE	Computer Aided Software Engineering
CFG	Context-Free Grammar
DB	Data Base
DBMS	Data Base Management System
DSL	Domain-Specific Language
DSM	Domain-Specific Modeling
DSML	Domain-Specific Modeling Language
EOL	Epsilon Object Language
EWL	Epsilon Wizard Language
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
GOF	Gang of Four
GPL	General Purpose Language
GUI	Graphical User Interface
IDE	Integrated Development Environment
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development
MDSE	Model-Driven Software Engineering

MVC	Model View Controller
MOF	Meta-Object Facility
OCL	Object Constraint Language
OCLE	Object Constraint Language Environment
OO	Object-Oriented
OMG	Object Management Group
PEC	Pattern Enforcing Compiler
PIM	Platform-Independent Model
PIT	Pattern Information Table
PMV	Pattern Language Semantic Verifier
PL	Pattern Language
PLA	Pattern Language Advisor
PLOP	Pattern Languages of Program
PLP	Pattern Language UML Profile
PLV	Pattern Language Verifier
PSV	Pattern Structural Verifier
PTV	Pattern Language Syntactic Verifier
PofEAA	Patterns of Enterprise Application Architecture
POSA	Pattern-Oriented Software Architecture
PSM	Platform-Specific Model
QOC	Questions, Options, and Criteria
UML	Unified Modeling Language
WFR	Well-Formedness Rule
XMI	XML Metadata Interchange
xUML	Executable UML
XSLT	Extensible Stylesheet Language Transformations

Chapter 1

Introduction

1.1 The Problem

The emergence of model-driven paradigm for software development has shifted the focus of software development from code-centric to model-centric. There are several approaches presented to the software community as model-driven approaches, e.g., Model-Driven Architecture (MDA) [Obj09b], Model-Driven Development (MDD) [Sel03, Sel06], Model-Driven Engineering (MDE) [Béz06, Sch06], and Model-Driven Software Development (MDSD) [SV06]. The ultimate goal in all these approaches is to be "Model-Driven." Hence, models are the main artifacts that drive software development in a model-driven approach [Béz06], and the quality of models has direct impact on the quality of software.

In this thesis, we select MDE as a representative of all the model-driven approaches. However, our focus is on the models that are used for the design of the software, i.e., *design model*. The use of model for designing software is not limited to the model-driven approaches, even in traditional software development paradigms, models are used extensively.

One of the approaches selected by the designers, in the hope of producing quality models, is to apply best practices that are already identified, documented, and introduced by the experts as *Patterns*. For instance, Booch says "All well structured object-oriented architectures are full of patterns" [GHJV95, p. xiii], or Larman says "Learning and applying patterns will accelerate your mastery of analysis and design" [Lar05, p. xxi]. This is because patterns are the documented knowledge of experts. When an expert finds out that a problem is occurring regularly, he/she may decide to form the solution as a pattern and introduce it to the others. In a short definition, "a pattern is a solution to a problem in a context" [GHJV95, p. 3].

When a collection of patterns is defined such that a) there is a starting pattern, b) there is a guidance on how to use one pattern after another, and c) the set of all patterns in the collection are sufficient to provide the design for a whole system, we name the collection a Pattern Language (PL) [A+77, Ale79]. If the designers decide to design a system based upon the patterns of a PL, they must have knowledge about how to apply an individual pattern correctly, how to put several patterns together (weave patterns) to make a correct combination of patterns, and how to ensure that a pattern combination is semantically correct.

Despite the ample discussion on the patterns in the software community, and the emergence of many pattern collections or pattern catalogs [HC07], the field of Pattern Languages is not as developed as the field of patterns. In an study, Booch [Boo09] has identified "a catalog of 1938 patterns, encompassing 54 pattern languages and 1884 individual patterns."

Building a design model in MDE based upon a PL is both recommended and widely accepted in the software community. In using a PL, two major issues are "pattern selection" and "pattern application" [GHJV95, p. 29]. These issues have direct impact on the quality of models. That means, selecting a wrong pattern or incorrect usage of a pattern could result in inconsistent design and therefore low quality software.

Not all the PLs have precisely defined rules governing the structure of individual patterns (*structural* rules), the possible relationships amongst patterns (*syntactic* rules), and the semantic of a pattern combination (*semantic* rules). For some PLs, pattern relationships are embedded into the lengthy texts of pattern descriptions. For instance, Patterns of Enterprise Application Architecture (*PofEAA*) [Fow02] consists of 51 patterns with relationships which are all explained in prose description. Hence, designers in general, and novice designers in particular, are vulnerable in making mistakes in pattern selection, pattern application, and pattern weaving. That means, *designing with patterns* [BHS07b, p. 248] is not an easy task, particularly for a novice designer.

Ensuring that the constraints of a pattern are respected frees a designer from the exigencies of implementation. Providing support for automatic verification of the models that have benefited from a PL will expedite the design process and results in better productivity. The ability to verify the use of a PL in a design model, results in better quality models (increasing the correctness of the model), faster development process, improved documentation, and improved consistency.

This thesis addresses the problem of verifying a *design model* which is built based upon the patterns of a PL. The problem is that patterns are not isolated islands. In designing software based upon the patterns of a PL, i.e., in *designing with patterns* [BHS07b, p. 248], the application of the patterns is not arbitrary. The PL may contain dozens of patterns with a variety of possible relationships between them: *uses, alternative, conflict*, to name a few. The designer must adhere to the relationships between patterns. If the designers do not understand the various interactions between patterns, they might select conflicting patterns [HAZ07]. Buschmann et al. [BHS07b, p. 121-134] argue and show by an example that the relationships between patterns help the designer create pattern-based designs where their quality is better than the designs built with isolated patterns.

Before presenting our solution, we must clarify that the verification, the correctness, and the consistency of a design model, are all with respect to the *rules* that describe the PL. By "verifying a design model" we mean we check a design model to find problems in applying the patterns of a PL. The problems are caused by violation of PL rules, *structural, syntactic*, or *semantic.* In other words we check the correctness and the consistency of the model from the PL point of view. By "obtaining a quality design model" we mean producing a model with improved quality, i.e., a model which is both correct and consistent. We verify a design model to find problems, then we fix the detected problems to obtain a quality model.

It should be noted that some researchers [Unh05, p. 14] correspond verification to only the syntactic correctness of software and models, and believe that dealing with semantic meanings is validation. However, in this thesis, we consider the checking of all three classes or *rules* as verification.

1.2 The Solution

As a solution to the aforementioned problem, we propose a process called Pattern Language Verifier (PLV). We believe that checking a model which is built using the patterns of a PL is similar to using a compiler [ASU86] for checking a source program which is written in a programming language. This similarity is the cornerstone of defining the PLV process. The idea of similarity between PL and a formal grammar is also pointed out by other researchers [NB02, HAZ07, Zdu07, BHS07a].

PLV is a verification process which exploits the idea of programming language compilers to detect the structural, syntactic, and semantic errors in a design model. Furthermore, PLV includes a module which helps the designer in fixing the problems either automatically or through guidelines and advices. We also propose a formalism for representing the rules of a PL. Characterizing the relationships between patterns is an open research problem [NB02].

The PLV process accepts a Unified Modeling Language (UML) design model as input and reports the structural, syntactic, and semantic errors in the model, considering the rules of the underlying PL. The PLV process includes four cooperating modules (phases): Pattern Structural Verifier (PSV), Pattern Language Syntactic Verifier (PTV), Pattern Language Semantic Verifier (PMV), and Pattern Language Advisor (PLA).

The process starts by PSV, which is responsible for detecting the structural errors that are found in the application of individual patterns. Then, the PTV detects the syntactic problems regarding the pattern combinations used in the model. The PMV finds the semantic problems in the design model, i.e., the inconsistencies between the detected patterns and the context of the design. The PLA reports the errors to the designer, gives guidelines on how to fix the problem, and, if wizards are available, repairs the problems automatically subject to the designer's request. A Design Rationale is also recorded by the PLA to show the automatic modification applied on the model. The information on the structurally correct patterns (detected by PSV) are recorded into a table called Pattern Information Table (PIT) which facilitates the work of the other phases.

As in writing a program in a programming language, where the programmer knows which keywords he/she wants to use and which language constructs is he/she using, in *designing with patterns* we suppose that the designer knows which patterns he/she wants to apply. This way we eliminate consideration of the task of pattern selection in PLV. For implementing this idea we utilize the UML profile mechanism [Obj05c]. This makes our process a **profile-driven** process. Both the designer and the process make use of the profile elements: stereotypes, tagged values, and constraints. Stereotypes are used to indicate which specific pattern is being applied and which model element is playing a role in a pattern. Tagged values are used to access the meta-information such as the level of expertise of the designer or the language used for implementation of the system.

1.3 Case Study

Our case study consists of two parts. The first part aims to validate the PLV process, i.e., to show how we can reach a PLV tool, given a PL. The second part intends to evaluate the obtained PLV, i.e., to show how the tool helps the designer in finding and fixing the design problems related to applying the patterns of the PL.

In the first part of the case study, as the PL, we select a subset of *PofEAA* [Fow02] consisting of 23 patterns. Since this PL lacks a set of precise rules that specify the structural, syntactic and semantic aspects of the language, we extract these from the *PofEAA* book. Then, the extracted information is transformed into formal rules. We define a UML profile for *PofEAA*, which makes the definition of a PLV for *PofEAA* possible. The defined PLV is then implemented as an extension for the ArgoUML [Tig09a] modeling tool. The resulting tool is called ArgoPLV.

In the second part of the case study, we use the ArgoPLV tool to model a sample web-based application: *online student registration system.* We choose two tracks for this part. The first track reveals how the ArgoPLV tool helps the designer in an interactive session with the tool, where a step-by-step design of the system is undergone. The second track shows the usefulness of the tool in verifying an existing model of the system and reporting the errors to the designer. The design before and after verifying with the ArgoPLV is investigated.

1.4 Contributions

To the best of our knowledge, PLV is the first work which addresses the problem of verifying a design model from the PL view. Most of the related work cited in Chapter 2 falls into the category of single pattern detection and those works do not focus on the PL aspects. There are two works close to PLV: Pattern Enforcing Compiler (PEC) [LSV05] and Zdun's work [Zdu07]. The former is an extension to a Java compiler which verifies the application of Gang of Four (GOF) [GHJV95] design patterns in the code. PEC only investigates individual patterns. It does not consider PL issues. The latter uses annotated PL grammars and design space analysis in systematic pattern selection. Zdun's work addresses both architectural patterns and GOF design patterns. This work provides a pattern selection mechanism; It is not a verifying approach, and it does not address the models directly.

Working on the formalisms for PLs in general, and particularly formalism for representing the pattern relationships, is an important area of research in software engineering. The rationale is that lack of formalism for PLs is an obstacle for providing tool support in pattern selection and application. We should note that patterns are not isolated islands, and considering patterns independently is not necessarily useful, even it may result in low quality designs, i.e. designs which are more complex and hard to maintain [SSRB00, p. 505], [BHS07b, p. 117], [Zdu07].

Addressing the quality in MDE is another important issue. Quality of a model, like any other quality, is not an absolute concept. Different people consider different quality attributes for a model. For instance, Selic [Sel03, Sel06] considers *Abstraction, Understandability, Accuracy, Predictiveness*, and *Inexpensiveness* as the characteristics of a *quality model*. Unhelkar's [Unh05] argues that a model should be *syntactically* correct, *semantically* meaningful, and *aesthetically* pleasing. Buschmann et al. [BHS07b, p. 131-132] see a model with *high pattern density* as a good design. After all, the tool assistance for quality assurance will help designers in finding the problems and checking the quality of the models.

The contributions of our research are as follows.

1. The PLV Process (See Section 3.5). This thesis moves the state-of-the-art in the Pattern Language Verification to the next level by introducing the PLV Process. The PLV is an improved version of the previously published ideas in [ZKB08, ZBK09]. For the first time, the idea of mimicking the tasks of the analysis phases of a compiler in order to check a design model is presented and organized as a process.

2. A formalism for representing a PL (See Section 3.3). This thesis contributes to the pattern formalization techniques by addressing all the three aspects of a PL: *structural, syntactic*, and *semantic.*

3. The *PofEAA* Advices (See Section 4.2). Extracting the advices from the book and classifying them into three groups, structural, syntactic, and semantic, is a useful source of knowledge for the designers who want to apply these patterns.

4. The formalized *PofEAA* rules (See Section 4.2). The advices are formalized using the formalism proposed in this thesis. These formalized rules pave the way for defining the constraints of the profile.

5. The *PofEAA* UML Profile (See Section 4.3). This is the first time that a profile is defined for a PL. The profile per se can be used by both the designers and the researchers.

6. The ArgoPLV (See Section 4.4). The ArgoPLV is a PLV for *PofEAA*, i.e, it is tool that verifies the application of *PofEAA*.

7. An exemplar session of ArgoPLV (See Chapter 5). This example shows *designing with patterns* of the *PofEAA* PL for a sample application: *Online Student Registration System.*

8. An MDE Road Map (See Section 2.1). The MDE road map presented in this thesis is an introductory review of MDE which discusses on the artifacts, the transformations,

the modeling tool, and the issue of "Quality in Modeling."

1.5 Organization of the Thesis

This thesis is organized as follows. Chapter 2 introduces the background knowledge and the related work. Chapter 3 describes the PLV process as a proposed solution to the problem. Chapter 4 shows how a PLV can be defined for a given PL. As a case study, a PLV is built for a subset of *PofEAA*. The PLV modules are integrated into ArgoUML [Tig09a] modeling tool, resulting to a tool named ArgoPLV. Chapter 5 shows the ArgoPLV in action to confirm that the ArgoPLV is applicable in a real world situation. The final chapter, Chapter 6, is dedicated to the conclusion, discussion, comparison to the related work, and future work. Appendix A, titled "ArgoPLV Artifacts," gathers together a comprehensive set of the artifacts that are generated during the process of building ArgoPLV tool. The appendix is a good reference for readers who are interested into applying the PLV process for a PL.

Chapter 2

Background and Related Work

This chapter introduces the basic background and the related work that help the reader understand the problem and the solution that are described in this thesis. Section 2.1 discusses the main idea of model-driven approaches, particularly Model-Driven Engineering (MDE), and addresses the issues regarding the quality control of models. A quick comparison to the traditional software engineering and Model-Driven Architecture (MDA) ends the section. In Section 2.2, we discuss Domain-Specific Modeling (DSM) and the related concepts, including Domain-Specific Language (DSL) and the UML Profile, and then a systematic approach for defining a profile is introduced. Section 2.3 elaborates on the concepts of pattern and Pattern Language (PL), and presents our accepted definition for the term PL. Section 2.4 illuminates the fact that "patterns are not isolated," by conveying the pattern relationships and the formalisms that are introduced by the pattern authors and the researchers. The work on pattern selection and/or pattern detection is discussed in Section 2.5. Section 2.6 introduces the Patterns of Enterprise Application Architecture (*PofEAA*) as a PL.

2.1 Model-Driven Engineering (MDE)

In most of the engineering disciplines, it is de rigueur to use models when designing a complex system. Since today's software systems are becoming more and more complex, benefiting from using models is inevitable [Sel03]. Despite the processes that are code-centric, in MDE models are the main artifacts which drive the development. The ultimate goal of MDE is to automatically generate programs from the corresponding models [Sel03]. Models are transformed from higher levels of abstraction to the lower levels such that finally they will become the deployable software.

In the software community, there is no clear presentation of the MDE as a process for software development. In this section, we present our literature survey on MDE as follows. In Section 2.1.1 we gather all the main concepts of MDE under an umbrella: "a road map for MDE." Then, in Section 2.1.2 to Section 2.1.4, we study the major parts of the road map, Artifacts, Transformations, and Modeling Tool respectively. Section 2.1.5 reviews the issue of quality control in MDE. In Section 2.1.6, we address the role of modeling in traditional software engineering, and in Section 2.1.7, the MDA is introduced briefly.

2.1.1 An MDE Road Map

Since MDE is not mature enough yet as a software process, there is no consensus on the life cycle and the artifacts in this process. As Bran Selic [Sel06] says "we are still in the infancy of this technological wave." We depict an MDE approach (road map) for software development as presented in Figure 1, with focus on the major artifacts that matter in MDE.

Figure 1 aims to highlight three major points about MDE:

- MDE is a model-centric approach.
- The software development is indeed a correct application of some transformations. That means developers transform artifacts from one level of abstraction to another level, until they obtain a working code.
- For applying MDE, we need a modeling tool to utilize automatic execution of transformations.

Note that this is not a complete MDE process, since it starts from the design, and the requirements are not considered as part of our MDE road map. This does not mean to underestimate the importance of the requirements analysis and specification. Instead, we assume that the designer already has some knowledge of the underlying business system.

2.1.2 Artifacts in MDE

In Figure 1, artifacts are shown by cubes. An artifact is a piece of information or the result of a transformation. As indicated in Figure 1, there are three main artifacts in MDE: Domain Knowledge, Model, and Code, which are discussed in the following.

Domain Knowledge The designer should have enough knowledge of the domain and some familiarity with patterns. Applying patterns is both essential and recommended in

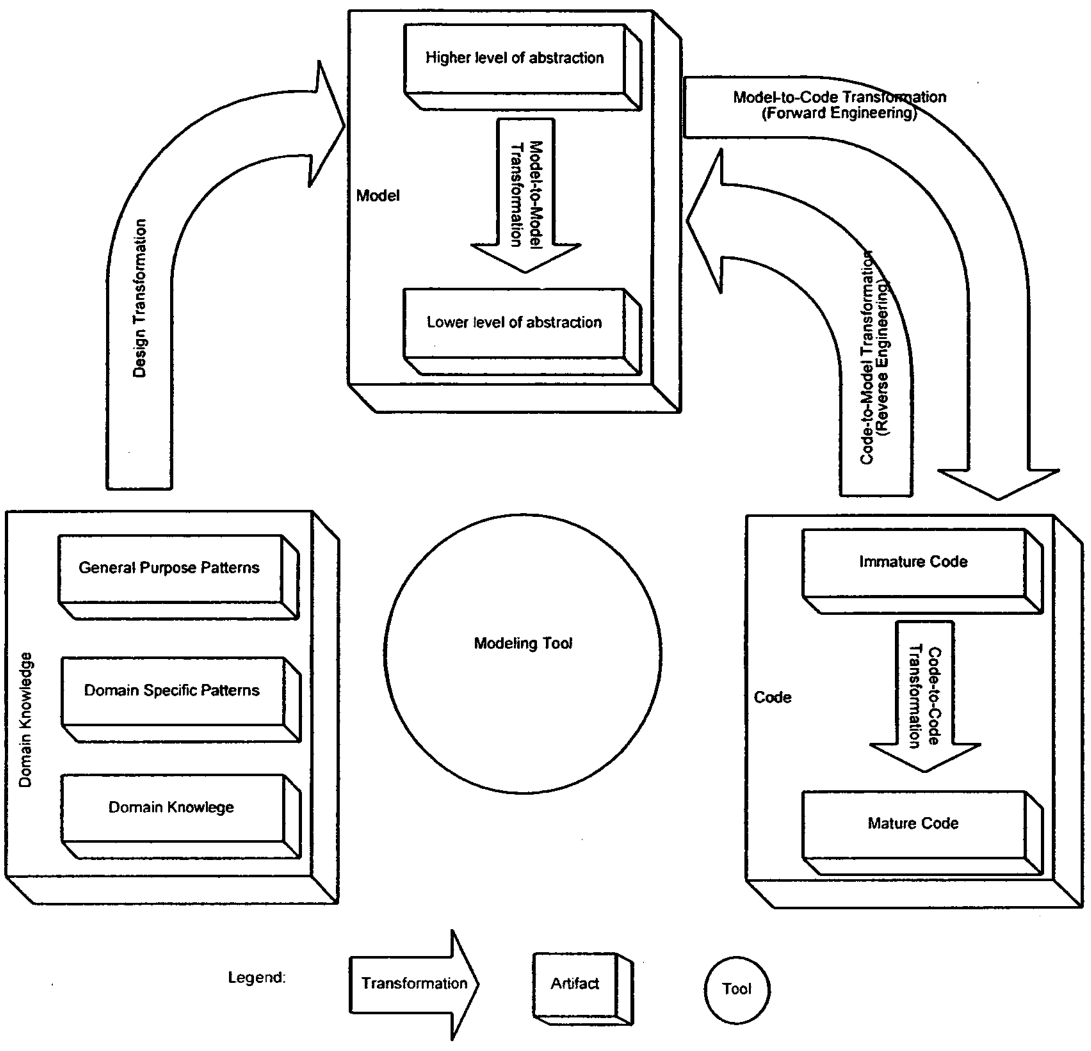

Higher level of abstraction
Model-to-Model Transformation
Model
Lower level of abstraction
Model-to-Code Transformation (Forward Engineering)
Code-to-Model Transformation (Reverse Engineering)
Design Transformation
Domain Knowledge
General Purpose Patterns
Domain Specific Patterns
Domain Knowlege
Modeling Tool
Code
Immature Code
Code-to-Code Transformation
Mature Code
Legend:
Transformation
Artifact
Tool

Figure 1: An MDE Road Map

software development using the MDE approach, e.g., Larman says "Learning and applying patterns will accelerate your mastery of analysis and design" [Lar05, p. xxi].

Patterns can be divided into two groups. General-purpose patterns include patterns and practices which are well known to the software community and their usage is recommended in most of the Object-Oriented (OO) software projects, GOF [GHJV95] design patterns and Larman's GRASP [Lar05] patterns, to name a few. Domain-specific patterns are applicable when the developer wants to work on a problem in a specific domain such as enterprise applications or telecommunication applications. As examples of domain-specific patterns, we can refer to *PofEAA* [Fow02] and Pattern-Oriented Software Architecture (POSA) [BMR+96]. We will discuss more about the patterns and PL in Section 2.3.

Model As it was mentioned earlier, models are the main artifacts in MDE. But what is a model? A basic definition is: *A model is a representation of a system.* Selic [Sel06] defines an *engineering* model as a representation of a system that hides some of the properties and highlights the ones that are of interest for the user. This hiding and highlighting means "a model is an abstraction" [Sel06].

Regarding the technical details of the system under development, models are divided into two levels of abstraction. Models in the high-level of abstraction deal with concepts that are more of interest of the user (customer) of the system and hide the technical details. Models in the low-level of abstraction contain technical and implementation details that are more attractive to the developers of the system.

In MDE, we start from a description of a business feature by building models which are at high level of abstraction. The final goal is to reach to models at lowest level of abstraction, i.e., an executable system. As we move in this path from start to end, our understanding of both the business goals and the system under development evolves. Therefore, it causes the models to evolve too. That means the models are more mature, accurate, and consistent [BIJ06]. Note that in Figure 1 we have distinguished between models at low-level of abstraction and the code.

Every model should conform to a metamodel. For instance, a set of books in a library can be represented as a relational model (a table which has columns such as ISBN, title, and author). Relational models conform to a relational metamodel which defines a relation (or table) as a set of attributes (or columns) with distinct names. Therefore, there are two important relations in MDE: representation and conformance. Simply put, the metamodel says what model elements we can have in our model and how these elements are arranged and related. Each element in the metamodel can be considered as a type for the model elements. Since there are a growing number of metamodels, there is a need for a meta-metamodel to be defined. A metamodel is said to conform to its meta-metamodel [Béz06]. Eclipse Modeling Framework (EMF) Ecore [Fou09a] and Object Management Group (OMG) Meta-Object Facility (MOF) [Obj06a] are two well known meta-metamodels.

There are many modeling languages, tools, and approaches. A modeling language can be mathematical, textual, or graphical. As an example of a mathematical language, Z [Spi92] is a formal specification notation based on the first order predicate logic and set theory. In the graphical modeling of OO software systems, Unified Modeling Language (UML) [Obj05b] is the dominant approach. "UML has become the universally-accepted language for software design blueprints" [Lar05, p. xix]. UML has an important role in the popularity of the application of models in software development [SV06, p. 3]. Our focus for high-level modeling in this research is on UML models. To narrow our previous definition of model, henceforth we consider the following definition for model: *A model is a set of related and consistent UML diagrams.*

To achieve the persistence and interchangeability of models, an interface format called XML Metadata Interchange (XMI) is provided by OMG [Obj05a]. Therefore, models are serialized in the form of XMI files by the modeling tools.

Code Code is the final artifact in any software development endeavor. In contrast to the traditional approaches of software development, in MDE the code generation is not merely the responsibility of the developers. That means, part of the code (and hopefully, all of it) might be generated automatically by the tool. The most productive form of MDE is when the process is fully-automated, i.e., the developers work only with the models and utilizing action languages, the code is automatically generated [Sel06]. This is the ultimate promise of the Model-Driven Software Engineering (MDSE) approaches that promote full code generation from the UML models, such as Executable UML (xUML) [MB02].

The code that is generated by currently available tools, is not mature enough to be considered as a running version of the software. Therefore, the code needs to be enriched by the complements added by the designer. The first version of the code which is automatically generated and mostly consists of code skeletons is named immature code in Figure 1. This code when added to by the developers and generates a working system is named mature code. Section "Model-to-Code transformations" will address this issue in more details.

2.1.3 Transformations in MDE

In Figure 1, Transformations are shown by arrows. Transformations are the distinguishing factor between the MDE and the traditional methods that use models only as sketches for the design [BIJ06]. Transformation is a mapping function that accepts an artifact as input and generates another artifact as output. By considering 'model' and 'code' as artifacts, there are four possible transformations: model-to-model, model-to-code, code-to-model, and code-to-code, which are discussed in the following. Some people consider code as a model with lower level of abstraction, hence, they define only one form of transformation: model-to-model transformation.

Transformations can be applied manually or automatically. In manual transformations, it is the developer's responsibility to investigate the input model and apply the modifications to it by adding, editing, or removing some model elements. Furthermore, the consistency of the resulting model is up to the developer. In automatic transformations, some transformation rules are defined to drive the changes, therefore the consistency of the output model is guaranteed. Transformation rules may be embedded into the modeling tool or maybe they are explicitly defined by the developer based on the domain-specific knowledge. As examples of these rules we can refer to profiles and patterns [BIJ06].

Model-to-Model transformation As it is clear from the name, in model-to-model transformation, a model is changed to another model. The source model and the target model could be instances of the same metamodel or different metamodels. When both source and target are from the same metamodel, there are two specific cases of model-to-model transformations: *refinement* and *refactoring*. In refinement transformations, a model is slightly changed to another model that better matches the desired system. Refinements can be done manually or automatically. Applying a pattern on a model is an example of automatic refinements, where the model elements are rearranged to satisfy the pattern requirements [BIJ06]. In refactoring transformations, the designer tries to reorganize the model and make it simpler based on some well-defined criteria.

Model-to-Code transformation Model-to-Code transformation is also called "code generation" or *forward engineering*. By this transformation, part of the code is generated automatically from the model. Code generation is one of the features that distinguishes MDE from the old paradigms of software development. Most of the modern modeling tools are capable of generating code skeletons for a given model. The ultimate goal of MDE is to reach the level of 100% automatic code generation. There is evidence [Dog07] that this dream does not seem to be elusive, considering the advances in the supporting technology [Sel06].

Code-to-Model transformation Since this transformation is the reverse of the Model-to-Code transformation, it is called *backward engineering* (or reverse engineering). By this transformation, changes in the code are automatically reflected in the model. If models are considered as the first class citizens in MDE [Béz06], then model should always be synchronized with the code. Otherwise, the model will be treated as a backup document which is deprecated soon after the system is delivered. Not many of the modeling tools are capable of performing backward engineering.

The union of *forward engineering* and *backward engineering* is called *round-trip engineering*. Having this feature, the developers are able to work on the model and the code concurrently. The idea is to keep the model synchronized with the code all the time during the system development [Sel06]. If the changes in the code are not reflected back in the model, then the maintainers face difficulties in maintaining the system.

Code-to-Code transformation This is not widely considered as a class of transformations in the MDE community. From our point of view, any change in the code can be called a Code-to-Code transformation, for instance, the refactorings [Fow99] that are applied into the code to make it simpler.

Design Similar to many of the software engineering approaches, design is a dominating step in MDE, since it relates to the modeling of solution space. As indicated in Figure 1, design is the outcome of a transformation, labeled "Design Transformation," which causes the existing knowledge of the system to take form and is revealed as a model.

Design can be divided into two levels: architectural design and detailed design. The former deals with the high-level design of software, such as the layering of sub-systems [Fow02, p. 2], and the deployment of modules. The latter is about technical design of each module or sub-system.

Most of the time, the design is based on instantiating well-known patterns, including general purpose and domain-specific patterns. Each pattern has an abstract template which contains some formal parameters that can be replaced by actual parameters. By pattern instantiation, the designer specifies the actual parameters for the parameters of the pattern. If the modeling tool is enriched with the pattern instantiation feature, like IBM Rational Software Architect (RSA) [IBM09b], most of the work is performed automatically.

2.1.4 Modeling Tool

To show the importance of the role of the modeling tool in MDE, in Figure 1, it is indicated by a circle that everything in MDE happens around it. In general, modeling tools are used for many purposes: to visualize, understand, and document existing systems, to create new designs, and to generate code for a design [LNH06].

However, in MDE, the modeling tool is anticipated to play a more prominent role by supporting tasks such as version control, process management, model driven testing, pattern definition and instantiation, checking the Well-Formedness Rules (WFRs) of models, import/export XMI format (serialization), detection and correction of inconsistencies between models, supporting UML profiles as an effective way of extending the UML metamodel, mapping between models, and both model-to-model and model-to-code transformations [Ken02].

We place emphasize on the role of modeling tools on improving the quality of the model. The tool assistance in finding and/or repairing the problems in models, fosters the quality control and quality assurance of the models.

2.1.5 Quality Control in MDE

Since models are the main artifacts which drive software development in MDE, quality assessment of models is an important issue. While people use models to enhance the quality of software, they must pay enough attention to the quality of models per se [Unh05]. Poor models will result in problems such as misunderstanding, wrong product, increase in test, and low quality system. Furthermore, the tool assistance for quality assurance is inevitable since merely manual inspection or review of designs is not enough [BCO05].

In the MDE process, the focus of quality checks must be on the models. There is no consensus on the answer to the question "what is a quality model?" Different people view the quality of a model from different aspects. Selic [Sel03, Sel06] considers a model to be a *quality* model, if it is "Abstract," "Understandable," "Accurate," "Predictive," and "Inexpensive." Unhelkar's [Unh05] looks at the quality of a model from three different aspects: "Syntax," "Semantics," and "Aesthetic." That means, model should be syntactically correct considering the modeling language rules, model should be semantically meaningful and consistent, and model should be aesthetically pleasing. From the patterns point of view, Buschmann et al. [BHS07b, p. 131-132] see "high pattern density" as a characteristic of a good design.

From the syntactical point of view, in UML documents, e.g., UML 2.0 Infrastructure [Obj05b], there exist some quality checks that are defined in the form of constraints or WFRs. WFR is a term used in the normative UML specification documents to describe a set of constraints that contributes to the definition of a metamodel element. WFRs are defined to help validate the abstract syntax and help identify errors in UML models. For instance, one WFR implies that "circular inheritance is not allowed in UML models." In addition to natural language, UML uses Object Constraint Language (OCL) [Obj06b] for expressing WFRs in a precise manner. However, the semantic and aesthetic checks, if described, are explained by natural language since they are contingent on the underlying domain of the model. Here is where Computer Aided Software Engineering (CASE) tools come into play and help designers in finding the problems and checking the quality of the models.

Checking Model Inconsistencies Egyed [Egy07] argues that some changes that the designers make in their models may have undesired side effects. That means, some changes may cause new bugs in the model, or they may make the model inconsistent. There are 34 consistency rules that are checked in the Egyed's work, e.g., "Rule 1: message name must match class method." Egyed has proposed an online, non-intrusive technique for fixing inconsistencies. It locates all choices for fixing inconsistencies, and identifies dependencies between inconsistencies. The technique is integrated into IBM Rational Rose [IBM09a], and is evaluated using 48 case studies.

Fuentes et al. [FQL$^+$03] have investigated the UML metamodel against the rules, constraints, and the WFRs defined by the UML standard, and have detected 450 errors. Many of these errors can be fixed easily, e.g., checking for empty names will solve about 300 errors.

Liu et al. [LEM02] have discussed that providing tool support for designers to find and repair problems in their designs, will help them improve the quality of the design. They have developed a production system named "Rule-Based Inconsistency Detection Engine" (RIDE) which helps the designers detect and resolve the inconsistencies in the UML models. RIDE is implemented in Java and can be integrated into modeling tools such as ArgoUML [Tig09a]. RIDE uses JESS [Lab09] to execute production rules. To detect inconsistencies in a given UML model, both the model and the inconsistencies must be converted into the production rules. Then the production system starts working by finding the inconsistencies and repairing them. In addition to general problems, RIDE can also be used to detect misuses of design patterns.

2.1.6 Modeling in Traditional Software Engineering

We consider two aspects of using models in traditional software engineering. From the one hand, there are several purposes for using UML models: making easier communication between people in a team, documenting the system and making the maintenance easier, helping in test case generation, to name a few. From the other hand, UML models are used in several phases of software development with different levels of abstraction. This usage varies from the early state of requirements specification (where use case models and activity diagrams are useful) to the further phase of architectural design (where package diagrams and deployment diagrams are used) [Unh05].

In traditional software engineering, modeling tools are used for drawing models. The models per se are considered as second priority artifacts, i.e., they are mostly prepared for design and documentation of the software. Since models are graphical and there is a belief that "one picture worth more than 1000 words," models are used vastly to ease the communication between developers, and to make maintenance easier. However, the extent of using models is not the same in all software development methodologies.

In *lightweight* methodologies (aka Agile processes [Coc06]), there is less focus on documentation (and modeling) than *heavyweight* methodologies. In Agile processes, modeling, especially in formal and tool supported format, has less value than working software [B$^+$09], and is done only if it is needed and if it helps in better understanding a design. In agile approaches, the focus is on making the design as simple as possible. The idea of agility in modeling has caused the invention of another terms such as "Agile Modeling" and "Agile Model Driven Development (AMDD)" [Amb02].

Quality Control of Models Several types of errors may exist in a model. First, the designer is vulnerable in making mistakes and creating wrong or low quality models in the design. Second, due to the fact that the semantics of UML is not strong enough (Fuentes et al. [FQL$^+$03] have reported 450 errors in the UML standard), there is possibility of inconsistencies between different models from different views. Third, the model may be not synchronized with the working code. This is plausible since models are not considered as the main artifacts, they are supportive documents that after the code is generated, there is no usage for them and they are going to be archived until a maintainer needs to refer to them to better understand the system.

The point is that the quality of model is as important as the quality of code. Even in less model-centric approaches, the models must be correct and high quality to be useful. The model should be checked against both the human errors, the inconsistencies that maybe remained in the model due to UML defects (inconsistencies), and the inconsistency with the working code.

In addition to the quality metrics in traditional software engineering, that root back to the code, e.g., Cyclomatic Complexity (CC) and Lines of Code (LOC), there are several OO metrics defined for evaluating the quality of models, e.g., Depth of Inheritance Tree (DIT), Number of Children (NOC), and Coupling Between Objects (CBO) [FP97]. However, further research is needed for finding quality models for design models.

Some quality models in traditional software engineering, e.g., ISO9126 [Int98], do not distinguish between the quality of an implemented software system and the quality of the description of the system. In code-centric approaches, the source code is per se the implementation and there is no sensible gap between them, however, in model-centric approaches, this gap (between the model and the implementation) is huge and therefore these quality models are not suitable for quality of UML models [Lan06].

There exist other techniques for quality control of a system, including the model and code, such as walkthroughs, inspections, and technical reviews [FW90]. In a walkthrough, a group of people gather together (including the producer) in order to give some comments about the product to the producer. Inspections are more formal practices in order to detecting and correcting defects in software artifacts. In a review, the product is examined by some individuals (other than the producer) in order to catch the defects.

Modeling tools can help in checking the syntax and semantics of the models. There are measurement tools, e.g., SDMetrics [sdm09], that analyze the design model (or the reverse-engineered code) using the OO measures and report potential problems to the designer.

2.1.7 Model Driven Architecture (MDA)

MDA is considered as an example of MDE vision. MDA was proposed by OMG in year 2000 [Obj09b], as a solution to the problems that were caused by constant changes in platforms. The proposal was based on two concepts, Platform-Independent Model (PIM) and Platform-Specific Model (PSM), and (automatic) generation of PSM from PIM. At first it was not precisely described how to generate PSMs from PIMs. Then, it was suggested that the PIM to PSM generation can be done by automatic model transformations [Béz06].

However, after a few years of research and practice in MDA, people are now considering more problems, other than separating PIM from PSM, that need to be solved. The separation and combination of concerns are currently major problems in development and maintenance of systems. PIM to PSM can be considered as a special case of a more general problem of separation of functional and non-functional requirements [Béz06].

MDA is considered as a perspective style of MDE. That means, models are defined precise enough adhering to specific semantics. Therefore, it is possible to apply consecutive transformations (mostly automated) on abstract models and obtain more concrete models. The final transformation will result in an executable system for a specific platform [BIJ06].

MDA has a lot in common with MDE, for instance both aim to move software development to a higher level of abstraction, but there are differences too. An important difference is that MDA is more restricted, due to the focus on UML [SV06, p. 4]. Creating a PIM is a crucial first step in the MDA process. The MDA tools should support the PIM to PSM generation vision and not just generate code from a class diagram.

2.2 Domain Specific Modeling (DSM)

Domain-Specific Modeling (DSM) can be viewed as a special case for MDE. In addition to the fact that in DSM models are still the main artifacts, we build a model of the system using the concepts that belong to a specific domain. That means, instead of working on low-level concepts, the designer deals with higher level of abstraction, resulting the increase in productivity. More productivity will be achieved if the domain is more specific [DSM09, PK02].

DSM consolidates several areas including DSL. A DSL is a language that is "tailored to a specific application domain" [MHS05]. A definition for domain is "An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area" [BRJ99]. In contrast to General Purpose Languages (GPLs) such as Java or C#, a DSL may therefore express a limited set of concepts and is suitable for a "specific class of problems" [Fow05]. Table 1 shows some of the DSLs.

DSL	**Application Domain**
BNF	Syntax specification
Excel	Spreadsheets
HTML	Hypertext web pages
LATEX	Typesetting
Make / Ant	Software building
MATLAB	Technical computing
SQL	Database queries and manipulation
VHDL	Hardware design

Table 1: Examples of domain-specific languages (adapted from [MHS05])

The idea behind DSL is that by using a large general purpose language, such as UML 2.0, we can not satisfy all the needs of the designers and users of a system. Especially, DSL helps non technicals to solve their problems without much help from technicals. A good example of a successful DSL is Excel which helps people in the domains such as business and finance [Béz06].

Models and DSLs both have strengths that urge us to use them together. For instance, Jouault and Bézivin [JB06] emphasize on the strong relation between DSLs and models. They define a DSL as "a set of coordinated models" and show how models can be used for defining the syntax and semantics of DSLs. As a proof of concept, the Kernel MetaMeta-Model (KM3) language is defined as a DSL for metamodel specification. KM3 is a meta-metamodel similar to MOF [Obj06a] or Ecore (the metamodel of Eclipse EMF [Fou09a] framework), however much simpler. While MOF has 28 classes and Ecore has 18 classes, KM3 has only 14 classes. Metamodels that are written based on KM3, can be easily converted to/from other formats such as EMFatic (Ecore format) or XMI (MOF format).

UML is the dominant metamodel in MDE, and OCL is a metamodel dependent language for writing constraints on UML models. However, the DSL approach encourages to use several small domain-specific metamodels instead of just using a single large metamodel such as UML. As a response to this need, Atlas Transformation Language (ATL) is defined as a metamodel-independent language that can be used for doing any kind of transformation on models. Especially, ATL can be used for checking models, also known as smell detection and refactoring [BJ06].

One of the major steps in applying a DSM is to have a Domain-Specific Modeling Language (DSML) [DSM09]. DSML is a technology required in MDE to be considered as a promising approach. There are several ways in defining a new DSML [Sel07]. One approach is to create it from scratch. A cost-effective approach is to select a more general language and customize (refine) it to the domain by specializing its general constructs. The UML Profile mechanism supports the *refinement* approach [Sel07].

2.2.1 UML Profiles

Different projects (and organizations) have different needs and use their own domain concepts [Ken02]. Therefore, it is needed to customize UML for specific domains. Fortunately, from the first day, UML was designed to be extendable and customizable [Sel07]. New modeling extensions can be introduced into UML by defining a UML Profile [Obj05c]. By defining a profile we can extend the UML metamodel with a set of new modeling elements [AN04, p. 10]. For doing domain specific modeling with UML, profiles are the recommended solution. UML profiles are extension mechanisms that allow you to tailor UML for specific areas such as Telecommunication. The idea of profile has been matured since its inception.

The first refinement mechanisms that were proposed in the UML were *stereotypes* and *tagged values* which were not defined very clearly and had not enough precision to be used for designing useful DSMLs. Then a package called *profile* is considered for holding all related stereotypes. In UML 2, the profile mechanism has received a lot of improvements in the rules and the definitions. Some of the improvements are: stereotypes can now have associations in addition to the associations of their base classes, profiles can be represented in XMI format, and applying (and un-applying) of a profile to a model is clarified [Sel07].

As one of the first documents introducing the profile idea, UML 1.4 Specification [Obj01, p. 2-74, 2-75] defines the extension mechanism and the profile concepts as follows.

> "The Extension Mechanisms package is the subpackage that specifies how specific UML model elements are customized and extended with new semantics by using stereotypes, constraints, tag definitions, and tagged values. A coherent set of such extensions, defined for specific purposes, constitutes a UML profile [...]
> A profile is a stereotyped package that contains model elements that have been customized for a specific domain or purpose by extending the metamodel using stereotypes, tagged definitions, and constraints. A profile may specify model libraries on which it depends and the metamodel subset that it extends.[...]
> Profiles are sometimes referred to as the 'lightweight' built-in extension mechanisms of UML, in contrast with the 'heavyweight' extensibility mechanism as defined by the MOF specification."

We found the following description of UML profile very brief and useful. It is provided by OMG in a page titled "Catalog of UML Profile Specifications" [Obj09a].

> "A UML profile is a specification that does one or more of the following:
>
> - Identifies a subset of the UML metamodel.
> - Specifies "well-formedness rules" beyond those specified by the identified subset of the UML metamodel. "Well-formedness rule" is a term used in the normative UML metamodel specification to describe a set of constraints written in UML's Object Constraint Language (OCL) that contributes to the definition of a metamodel element.
> - Specifies "standard elements" beyond those specified by the identified subset of the UML metamodel. "Standard element" is a term used in the UML metamodel specification to describe a standard instance of a UML stereotype, tagged value or constraint.
> - Specifies semantics, expressed in natural language, beyond those specified by the identified subset of the UML metamodel.

- Specifies common model elements, expressed in terms of the profile."

Some examples of the UML profiles listed in the catalog [Obj09a] are: UML Profile for Enterprise Application Integration (EAI), UML Profile for Systems Engineering (SysML [Sys09]), and UML Testing Profile.

List and Korherr [LK05] have presented "A UML 2 Profile for Business Process Modeling," which is claimed to be more comprehensive than the previous profiles on business process modeling. Note that in UML 1.4 Specification [Obj01, p. 4-9] a "UML Profile for Business Process Modeling" is introduced as an example.

Ziadi et al. [ZHJ03] have introduced an introductory work towards defining a UML profile for software product lines.

Kobryn [Kob04] has discussed the good, bad and ugly sides of the UML 2.0 and addressed the future of modeling. He refers to the Systems Modeling Language (SysML [Sys09]) as good sample of extending UML 2.0 towards a modeling language for systems engineering. He emphasizes on an important point that profiles are not only for extending the UML, but also they can be used for subtracting features from the language.

2.2.2 A Systematic Approach for Defining UML Profiles

While designing a UML profile does not seem to be a difficult task, it should be done with care. Mostly, a UML profile is just a set of possible stereotypes and tagged values. Therefore, a profile is facilitating domain specific modeling. In the course of design, you can annotate the model elements with the defined stereotypes. However, the importance of the role of those stereotypes becomes clear when we notice that they are defining a new language that we can work with as we model our domain [Unh05].

Bran Selic [Sel07] has addressed the issue of low quality profiles. Selic believes that lack of enough material and guidance for designers, on how to create a good profile, is the cause of these poor quality profiles. As a solution, Selic has proposed a systematic approach for defining a profile. In fact Selic's approach is targeting the design of DSLs using the UML profile mechanism. Selic's approach is separated into two steps.

1- Defining the domain metamodel At the first step, without considering the UML metamodel, we define a domain model of the DSL that we are designing. This domain model is in fact the metamodel of our language. This metamodel consists of all fundamental concepts from the underlying specific domain, the relationships between those concepts, the constraints (WFRs) for valid models, the notation of the DSL, and the semantics of the

DSL. It is wise to express the abstract syntax of the language using MOF and to write the profile constraints in OCL, since UML metamodel is also defined using MOF, and OCL is supported by many UML tools.

2- Mapping the domain metamodel to UML metamodel The second step is to map each of the concepts in the domain model into one of the appropriate base classes in the UML metamodel. Then for each concept, one stereotype should be defined. It is possible that some stereotypes need to be considered as the specialization of other abstract ones. The steps should be done carefully in order to prohibit inconsistencies or conflicts between the attributes, associations, and constraints of the domain concepts with the corresponding UML meta-class.

2.3 Pattern Languages (PLs)

Despite the ubiquity of the concepts Pattern and Pattern Language (PL) in software engineering, there is no formal definition for them. Due to the fact that the "Pattern Language" concept plays a key role in this thesis, this section is dedicated to provide a clear definition for Pattern and PL. We start from the architecture area, where the story started, then we move to the software area to review the definitions given by the experts in the field and to give our definition.

2.3.1 Pattern Languages in Architecture

The terms "Pattern" and "Pattern Language" were first coined in late 60's by Christopher Alexander [A+77, Ale79], an emeritus professor of architecture at the University of California at Berkeley. Amongst many books written by Alexander, there are two books which have influenced software community a lot: "A Pattern Language: Towns, Buildings, Construction" [A+77] and "The Timeless Way of Building" [Ale79]. The former is a collection of 253 *inter-related* patterns for architectural design elements that, all together or a subset of them, form a language. The latter shows a systematic way for using these patterns in designing part of the environment.

In 2000, Alexander founded the `patternlanguage.com` company to promote collaborative working between people, builders, and architects to build good buildings. On the `patternlanguage.com` web site the story of how this name is selected for the company is explained. The following is an excerpt from that story which summarizes the PL concept:

> “Once upon a time, we wrote a book called A Pattern Language and that is how we got our name. [...] The new idea in the book was to organize implicit knowledge about how people solve recurring problems when they go about building things. [...] Patterns are easy to remember and set out as if-then propositions. [...] We were surprised though, when we found out computer programmers liked it, because it was about building not programming. But the programmers said, “this is great, it helps think about patterns in programming and how to write reusable code that we can call upon when we need it.” [...] Now a pattern language is about patterns being like words. They stay the same but can be combined in different ways like words in a sentence. They can be used as in a network where one will call upon another (like a neuron network). When you build something you can put patterns together to form a language. So a language for your house might have patterns about transitions, light, ceiling height, connecting the second floor to the ground. [...] But what we're working most hard at is writing sequences. Now a sequence is something that looks very very simple and is actually very very difficult. It's more than a pattern; it's an algorithm about process. But what is possible is to write sequences so that they are easy. You follow the steps in a sequence like you follow the steps in a cooking recipe. [...] A sequence is figuring out which decision has to come first and getting it right and then moving to a second decision. [...] An architect who uses such a sequence, can do better and more beautiful work. [...] A lay person can make a design, at least in a simple form, where previously it was assumed that only architects and engineers could make designs.”

From the above text, we find the following important facts about the patterns.

- Patterns are tools for organizing the implicit knowledge that people use for solving a recurring problem.
- This solution knowledge is normally organized as if-then rules.
- PL is like a network of patterns that one can call upon another.
- We need to write sequences of patterns that act as cooking recipes and are easy to follow for a lay person.
- Even an expert may use a sequence and build better designs.

2.3.2 Pattern Languages in Software

Alexander in [A+77] says: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." This definition is summarized by software gurus, as the definition of *pattern* [NB02]:

"A pattern is a solution to a recurring problem in a specific domain."

In the software community, there are a few works which have selected the format used in "A Pattern Language: Towns, Buildings, Construction" [A+77] for defining a PL. For instance "A Pattern Language for Writing Patterns" [MD97] is an article which contains a set of tightly related patterns such that selecting one pattern leads the user to another pattern. And the set of proposed patterns is a complete list that enable a person to perform a pattern writing project.

Patterns have played an important role in software development in general, and in object-oriented approach in particular. Kent Beck and Ward Cunningham [BC09] are the first persons who applied patterns to software [Nob98b]. The idea was then popularized by the publication of the seminal book on design patterns known as the "Gang of Four" (GoF) design patterns [GHJV95]. "A growing number of people consider design patterns to be a promising approach to system development, [...] especially in object-oriented systems" [Zim95].

The GOF is used by many software experts and is cited by many researchers, e.g., as of the day of this writing, its citation count on ACM is 1984 and on Google Scholar is 17100; more than 500,000 copies of the book is sold and it is translated into more than 13 languages, and with 243 reviewers in amazon.com, it ranks 2^{nd} in the Software Engineering Bestsellers category.

2.3.3 Forms of Writing Patterns (Pattern Forms)

Software experts have defined (discovered) hundreds of patterns as solutions to recurring problems in software design. For describing the structure of the patterns, each pattern author has his/her own *pattern form*. There is no consensus on the structure and elements of a *pattern form* between different pattern authors. In a survey on pattern collections, Henninger and Corrêa [HC07] have concluded that "Almost every pattern collection we surveyed used a *different* pattern form." They claim that "Lack of Standard Pattern Forms" is one of the challenges for federating software patterns [HC07]. But, it should be noted that we can not expect to have only ONE *pattern form* that fits the needs of every PL. That means, there should be several standards for *pattern forms.*

A *pattern form* consists of several items. The authors of GOF book stress that, in general, a pattern needs four essential elements: the pattern name, the problem, the solution, and the consequences [GHJV95, p. 3]. The GOF design patterns' form contains the following items (sections): Pattern Name and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, and Related Patterns [GHJV95, p. 6-7].

Some software pattern authors have adopted the *pattern form* introduced by Alexander in [A+77]. For example, the form used in POSA-4 [BHS07a, p. 48] includes: name, context, main (which includes problem statement, forces, solution instruction, solution sketch, solution structure and behavior), and solution consequences.

The *pattern form* used in *PofEAA* includes eight items as follows [Fow02, p. 11].

1. The name of the pattern: Pattern names are crucial since they they create a vocabulary to be used by designers when they communicate.
2. The intent: The intent is a short description of the pattern.
3. The sketch: The sketch is a graphical representation of the pattern, mostly as a UML diagram.
4. A motivation problem: A sample problem that the pattern can solve.
5. How It Works: This is in fact the solution to the problem. It explains the implementation issues and the variations that can be considered. For some patterns, UML class diagrams or sequence diagrams are presented as an aid to explain them.
6. When to Use It: This item shows the justifications about why to use this patterns comparing to others. (In some pattern forms this is called *forces.* Forces are the factors such as cost and performance.)
7. The Further Reading: This is a reference to the information that may help the reader better understand the pattern.
8. The Examples: There are one or more examples on how to implement the pattern in programming languages Java or C#.

One important aspect of a pattern is its name, because by documenting patterns and the relationship amongst them, in fact the pattern author is defining a language, called Pattern Language (PL), that could be used by designers in developing new software systems [Ber94]. In other words, pattern names play a crucial role in a PL, because designers can use those names as a vocabulary that helps them to communicate more effectively [Fow02, p. 11].

However, a PL is not only a collection of patterns. To emphasize on the dependencies between patterns, Alexander [A^+77] expresses that "the link between the patterns are almost as much a part of the language as the patterns themselves." Also, earlier (See Section 2.3.1) we read about Alexander's idea on Pattern Language as "patterns [...] can be combined in different ways like words in a sentence." As an analogy, we can consider each pattern as a recipe for a solution, therefore, a PL is a set of recipes for a whole system.

2.3.4 Pattern Language Definition

There is no consensus on the definition of a Pattern Language (PL) in software community. In the followings we quote viewpoints of several software experts on PLs, then, we adopt our definition of PL.

"A pattern language defines a collection of patterns and the rules to combine them into an architectural style. Pattern languages describe software frameworks or families of related systems" [Hil09b].

"A PL is a set of patterns that guide an architect through a design. Each pattern is a description of a solution to a problem using other patterns that occur in the system" [Ber94].

"A pattern is a recurring solution to a standard problem. When related patterns are woven together they form a "language" that provides a process for the orderly resolution of software development problems. Pattern languages are not formal languages, but rather a collection of interrelated patterns, though they do provide a vocabulary for talking about a particular problem" [SFJ96].

"A *pattern language* is a collection of interrelated patterns organized into a coherent whole, which provides a detailed solution to a large-scale design problem" [Nob98b].

"One of the key advantages of a pattern language over a standalone pattern is its ability to guide the reader to the solution of a complex problem by leading them from one pattern to another. Stand-alone patterns have to work harder to establish their relationships" [MD97].

"The idea behind a pattern language comes again from Alexander. The idea is that you have a body of patterns with a structure that leads you from pattern to pattern. You begin with (usually) some very strategic patterns. each pattern leads you to a point where you have to decide to apply other patterns. A pattern language has a flow that connects the various patterns. [...] I look at pattern languages as a structuring mechanism" [Fow06].

One of the most recent and comprehensive discussions of PLs is presented in POSA-5 by Buschmann et al. [BHS07b]. They believe that a PL is "A Process and a Thing." The 'process' part tells the designers *how* to solve a problem, and the 'thing' part tells about *what* are the concrete solutions that can be solved by the PL. The 'thing,' is a specific kind of software system created by the 'process.' Missing any of the 'process' or 'thing' parts, causes the PL not to be able to systematically resolve software development problems [BHS07b, p. 260].

Buschmann et al. [BHS07b, p. 260] argue that the following definition is both acceptable for a PL and is in line with the pattern community, however, it should be noted that the focus of this definition is on the 'process' concept.

> "*A network of tightly interwoven patterns that defines a process for systematically resolving a set of related and interdependent software development problems*" [BHS07b, p. 260].

In this thesis, we select the above definition for a PL.

2.4 Pattern Relationships

2.4.1 Patterns are not Isolated Islands

Patterns are not isolated islands. Considering patterns independently results in low quality designs, i.e. designs which are more complex and hard to maintain [BHS07b, p. 117]. Patterns can be used individually, however stand-alone patterns are able to solve only tiny problems because they do not consider larger contexts. Hence, one should note that using single patterns does not help building real-world software systems in an efficient manner. In order to increase the power of patterns, pattern authors should show how the patterns can connect, complement, and complete each other to make a PL. The resulted PL then can be used in designing high quality systems [SSRB00, p. 505-506].

Even in the pattern resources that have not focused on the PL aspects, e.g., in GOF, we can see indications of considering the dependency between patterns. This indications varies from a graphical map to a prose description of pattern dependencies in a dedicated field in the *pattern form*, e.g., the Related Patterns field in GOF patterns.

In addition to the relationship between the patterns of a PL (inter-collection relation), patterns of different PLs might also be dependent on each other (intra-collection relation). Considering the importance of inter-collection relationships, the pattern community sees that ongoing work on patterns is focused more on synthesis and connection than introducing new patterns. Examples are *PofEAA* [Fow02] with 51 patterns, POSA-4 [BHS07a] with 114 patterns (with connections to more than 180 other patterns), and Grady Booch's project [Boo09] on defining a "Handbook of Software Architecture" that so far has identified more than 1800 patterns including 28 PLs [BHS07b, p. 132].

Henninger and Corrêa [HC07] have addressed the problem of growing number of patterns and pattern collections. Based on a survey on available pattern collections, they promote utilizing Semantic Web technology for formal specification of pattern collections. In a project named "An Ontology-Based Infrastructure for Creating Software Pattern Languages," they set up the Semantic Framework for Patterns (SFP) web site [Uni09b]. As indicated in the web site, the goal of the project is "to create a repository representing the many facets of software patterns." The site is now open to public to add or edit the information about the existing pattern collections. As of date (20 April 2009) there are 234 pattern collections containing 2935 patterns recorded in the SFP web site.

2.4.2 Pattern Relationships and Quality of Design

In order to see how considering the relationships between patterns during design, affects the quality of the design, consider the Request-Handling framework example which is borrowed from [BHS07b]. The problem is "developing an extensible request-handling framework that helps to transform service requests from clients into concrete method invocations on an application" [BHS07b, p. 123].

In the first approach, we do not consider any relationship between patterns and naively treat them as isolated islands. Our first task is to find a pattern that solves the problem of "objectifying" the requests of clients. One solution is the COMMAND pattern. Then, the second task is how to handle the commands by a central component. We find the COMMAND PROCESSOR pattern as a solution and integrate it to the existing architecture. Third, for supporting "undo" or the rollback of the actions, we select the MEMENTO pattern. But we need a separate Caretaker class to relate the command with the Memento. Forth task is "logging" the requests, for which, the STRATEGY pattern is selected and is glued to the existing architecture by a LoggingContext object. And the final task is to support compound commands. The COMPOSITE pattern addresses this feature, which for gluing it to the existing design, we insert it between the COMMAND PROCESSOR and the COMMAND patterns. Figure 2 shows the resulted design consisting of five patterns. Clearly, this design is not a good design due to complexity, and difficulty in understanding and maintaining [BHS07b, p. 123-128].

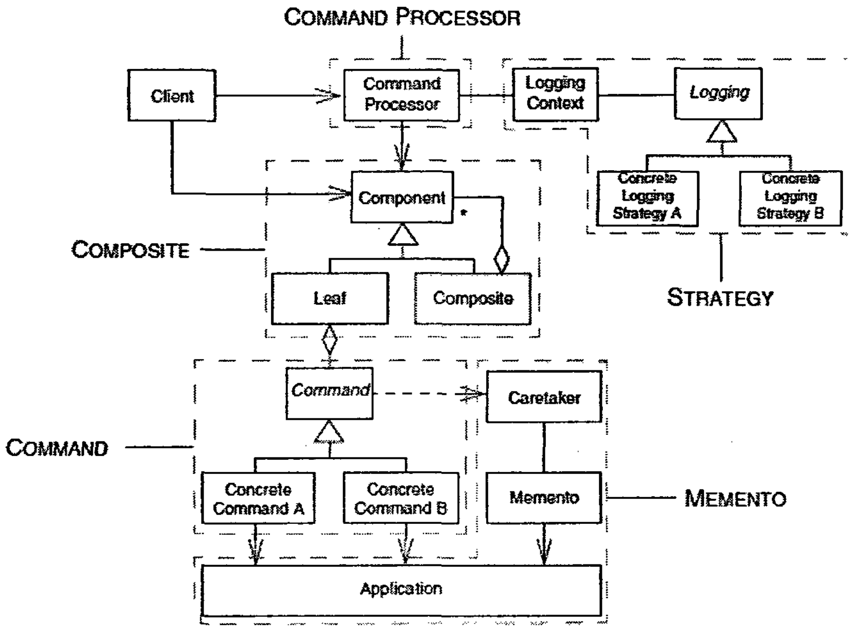


Figure 2: Design of Request-Handling Framework with Isolated Patterns [BHS07b, p. 128]

In the second approach, we consider the same set of patterns but with bearing in mind the possibility of interweaving the patterns together. For example, COMMAND and COMPOSITE patterns can be combined. After several refactorings, we reach to the structure given in Figure 3 which is much easier to understand and maintain [BHS07b, p. 129-131].

Comparing the above two designs, we see that the second one, which is a better architecture, has a high density of tightly integrated patterns. Actually, this feature, e.g., high pattern density, is a characteristic of a good design. In simple terms, the pattern density is defined as the number of patterns per number of classes. However, compressing many patterns in as few classes as possible is not equivalent to having better design. That means, weaving patterns together must be done accurately and precisely [BHS07b, p. 131-132]. In connecting patterns, in addition to the roles of their components, we should also consider the relationships between those components.

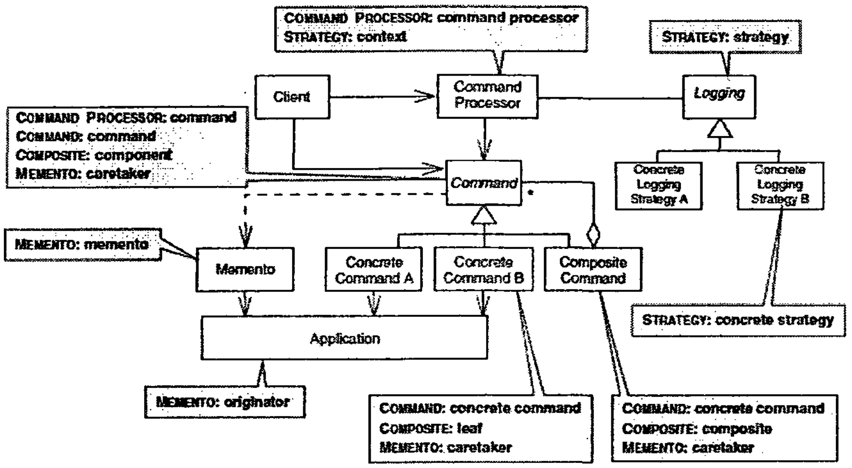


Figure 3: Design of Request-Handling Framework with Interwoven Patterns [BHS07b, p. 131]

2.4.3 GOF Pattern Relationships

For GOF design patterns, the "Related Patterns" field of the *pattern form* briefly talks about how patterns reference each other. The relationships between all 23 patterns are shown in Figure 4.

Zimmer [Zim95] has studied the relationships between GOF design patterns, and has categorized them into three categories: "uses," "is similar to," and "can be combined with." Based on this classification, a new diagram is proposed for the relationships between the GOF patterns. This diagram is shown in Figure 5.

Zimmer has concluded two important points [Zim95]:

- "Applying design patterns requires a fair knowledge of both single design patterns and their relationships."
- "Tool support is needed to apply design patterns to really large applications."

2.4.4 James Noble's Pattern Relationships Scheme

By studying several pattern collections and the way that these collections have documented the relationships between patterns, James Noble [Nob98a] has found that there is no standard for describing pattern relationships. He says "Unfortunately, each pattern text book or catalog describes relationships between patterns using its own idiomatic classification of these relationships" [Nob98a]. To address this problem, Noble has proposed a classification scheme for the relationships between patterns. His scheme consists of three *primary* relationships and nine *secondary* relationships as indicated in Table 2.

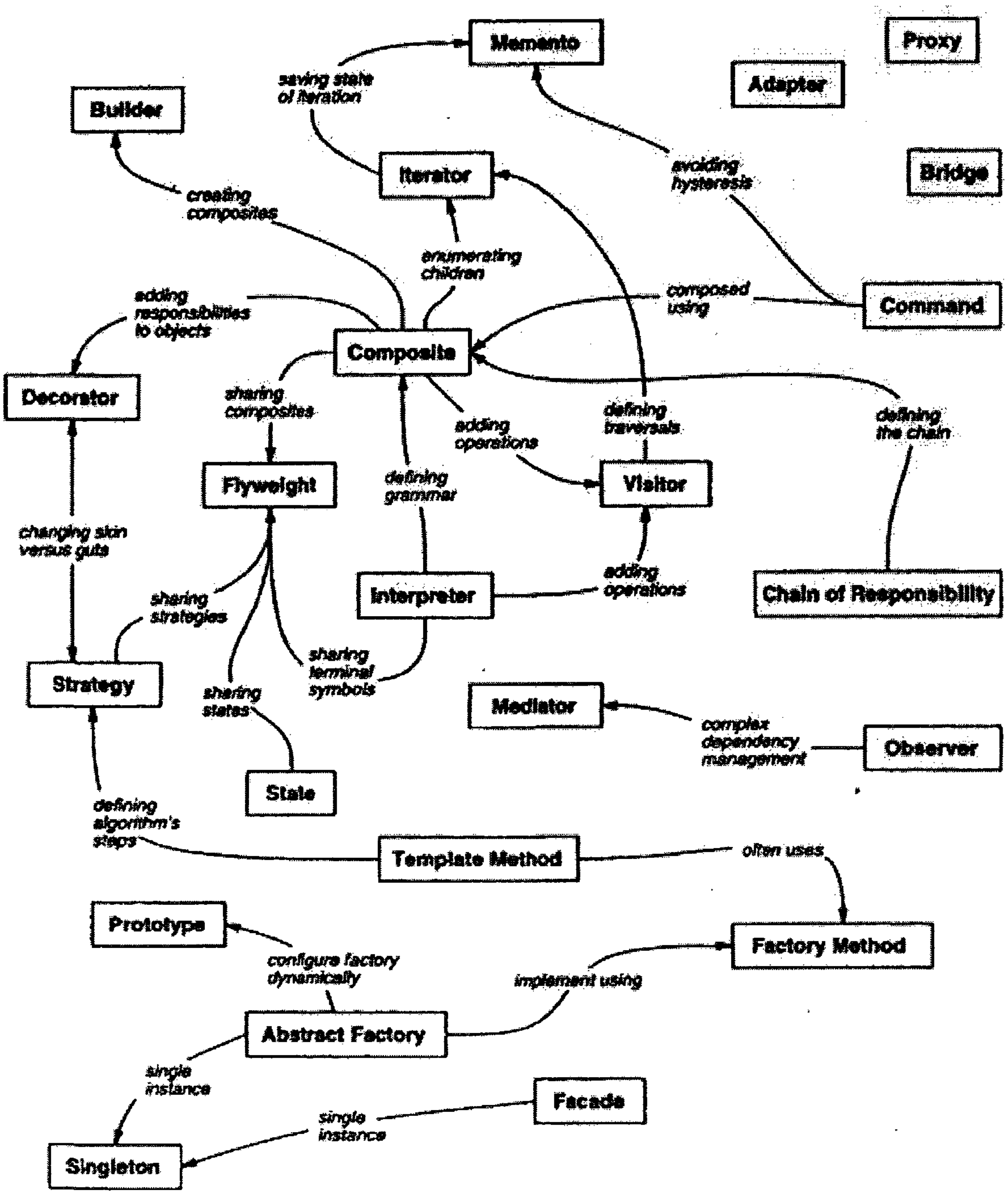


Figure 4: Relationships Between GOF Design Patterns [GHJV95, p. 12]

Primary relationships are the ones that are widespread in the patterns literature, act as the basis for describing other patterns, and their definitions are straightforward. The *secondary* relationships are the ones that can be be expressed in terms of the primary relationships.

Uses relationship shows how a large pattern may be composed of small-scale patterns.

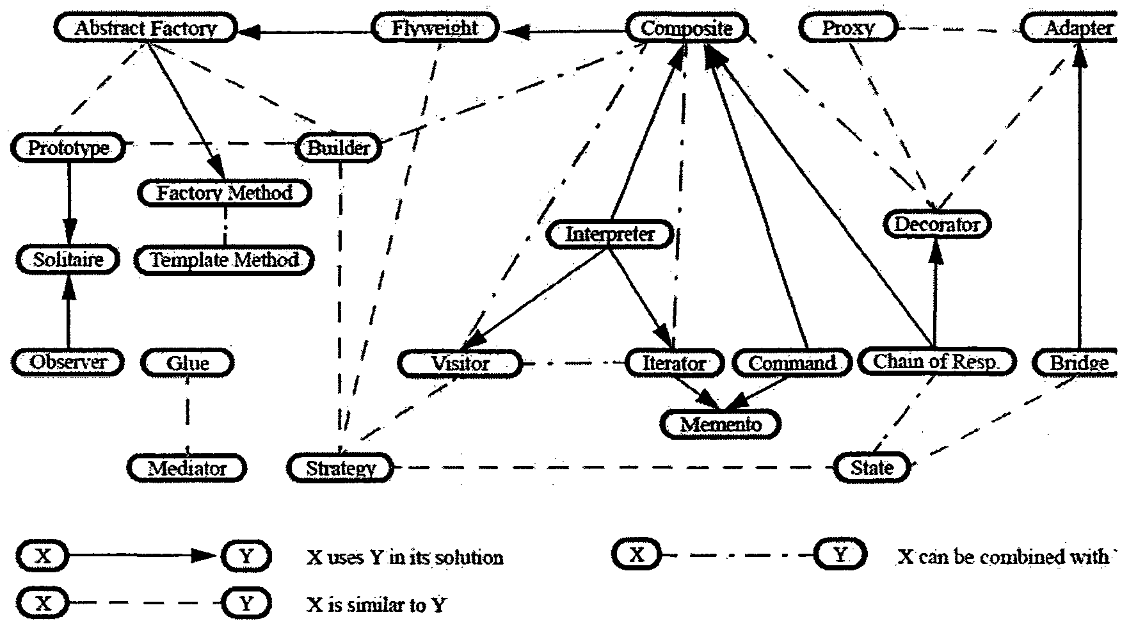


Figure 5: Relationships Between GOF Design Patterns Proposed in [Zim95]

Table 2: Classification of Pattern Relationships [Nob98a]

Primary Relationships	
Uses	One pattern uses another pattern
Refines	A specific pattern refines a general pattern
Conflicts	A pattern addresses the same problem as another pattern
Secondary Relationships	
Used by	smaller pattern is used by a larger pattern
Refined by	general pattern is refined by a specific pattern
Variant	variant pattern refines a more well-known pattern
Variant Uses	variant of one pattern uses another pattern
Similar	pattern is similar to another pattern
Combines	Two patterns combine to solve a single problem
Requires	pattern requires the solution of another pattern
Tiling	pattern uses itself
Sequence of Elaboration	sequence of patterns from the simple to the complex

This relationship is mostly documented in the "Related Patterns" or "See Also" section of a *pattern form*. For instance, in GOF patterns, Observer *uses* the Mediator pattern for coordinating multiple objects updates, or Mediator *uses* Singleton for preventing duplication of mediators. The *uses* relationship can be interpreted as the composite relationship in the OO world.

Refines relationship shows how a pattern is an special case of another one. This relationship is mostly implicit in the description of patterns, i.e., there is no dedicated field in the *pattern form* that shows which pattern is a refinement of the other one. For instance, in GOF patterns, Factory Method *refines* Template Method. The *refines* relationship can be seen as the inheritance relationship in the OO world.

Conflicts relationship exists between the patterns that are solutions to the same problem, but they are mutually exclusive. Reviewing the "Related Patterns" or "See Also" section of a *pattern form* would help in finding patterns that are conflicting with the current pattern. For instance, in GOF patterns, Decorator *conflicts* with Strategy, because both are solution to the problem of modifying the behavior of other objects. It is a good practice to investigate all the conflicting patterns while solving a problem, but only select one of them.

In another paper, Noble and Beedle [NB02] have listed some of the open research problems regarding patterns as follows. How can we differentiate patterns that are structurally similar (e.g., Strategy and State)? How can we know that one pattern can be a solution to more than one problem (e.g., Proxy)? How can we know that one pattern can have distinctly different variant forms (e.g., Adapter)? How can several different patterns have the same name (e.g., Prototype)? How can we characterize the relationships between patterns?

2.4.5 Pattern Language Grammars

Using the vocabulary metaphor for the patterns of a PL leads to the grammar metaphor for the rules that dictate the correct sentences of the language. That means, "each pattern sequence can be viewed as a properly formed sentence in a pattern language" [BHS07b, p. 281], and "The union of all pattern sequences supported by a pattern language can thus be understood as its full set of grammatically correct sentence forms" [BHS07b, p. 282].

But, the sequences only show the results of applying the grammar rules, not the rules per se. That means, the rules are implicit in the sequences. For making the syntactic (grammatical) rules explicit, there exist two approaches. First, to integrate the rules into the descriptions of constituent patterns. Second, to use a formal notation for describing the grammar rules. The drawbacks of the first approach are vagueness and ambiguity of the rules [BHS07b, p. 282]. The disadvantage of the second approach is that expressing the grammar of a large PL in a formal notation is difficult [BHS07b, p. 285]. Hence, most of the PL authors have preferred the first approach, i.e., to present the syntactic rules in prose, interwoven with the pattern descriptions [BHS07b, p. 284].

Following are alternative formal notations, that can be used for representing the grammar of a PL, along with examples given for the Request-Handling framework introduced in Section 2.4.2.

BNF Notation The BNF [Knu64] notation which is widely used for writing the syntax of a programming language, can also be tailored for the PLs. In the following, we present our idea of defining a grammar for a PL. This definition is inspired by the seminal works [ASU86, Lin06] in the field of formal languages and automata theory.

A grammar for a PL is a quadruple $G =< N, T, S, P >$ such that:

- N is the set of non-terminals of the PL. A non-terminal is a temporary variable which will finally be replaced by a sequence of patterns. We suppose that non-terminals (variables) are represented by the words with lower-case letters.
- T is the set of terminals (patterns) of the PL. In other words, T is the alphabet of the language. Terminals (patterns) are shown with capitalized words.
- $S \in N$ is the starting variable of the grammar. If it is not explicitly specified, the variable that appears first is considered as the starting variable of the grammar.
- P is the set of production rules that dictate how a sequence of patterns can be built.

The production rules of the grammar are in the form $A \rightarrow \alpha$, where $A \in N$ is a variable and $\alpha \in (N \cup T)^*$. In making α, the operation '.', is a binary operation that shows a dependency from the left operand to the right operand. The operation '*' means any number of applying '.' operation. The alternative operation '—', is a binary operation and shows a choice between either of the two operands. Operator '.' has priority over '—' and parentheses are used for grouping. The terminal λ means null or nothing. As an example, in the following a grammar is given for the Request-Handling PL described in [BHS07b, p. 283].

---

start $\rightarrow$ *COMMAND . EXPLICIT INTERFACE . temp*1 | λ
*temp*1 $\rightarrow$ *MEMENTO . temp*2 | *COMPOSITE . temp*3 | λ
*temp*2 $\rightarrow$ *COMPOSITE . temp*4 | λ
*temp*3 $\rightarrow$ *COMMAND PROCESSOR . COLLECTION FOR STATES .*
STRATEGY . NULL OBJECT | λ
*temp*4 $\rightarrow$ *MEMENTO* | λ

Figure 6: A BNF Grammar for the Request-Handling Pattern Language, Adopted from [BHS07b, p. 283].

POSA-5 Notation A new notation is introduced in POSA-5 [BHS07b, p. 282] inspired by the BNF. Following is a grammar for the Request-Handling framework using this notation.

$\oslash$ stands for starting state, $\rightarrow$ shows the mandatory sequence, $\xrightarrow{o}$ denotes the optional sequence, — is for alternation, and () is used for grouping.

$\oslash \xrightarrow{o}$ *(COMMAND* $\rightarrow$ *EXPLICIT INTERFACE* $\xrightarrow{o}$ *(MEMENTO* $\xrightarrow{o}$ *COMPOSITE* $\xrightarrow{o}$ *COMMAND PROCESSOR* $\rightarrow$ *COLLECTIONS FOR STATES* $\rightarrow$ *STRATEGY* $\rightarrow$ *NULL OBJECT)* | *(COMPOSITE* $\xrightarrow{o}$ *MEMENTO))*

Graphical Notation There are several graphical notations that can be adopted for representing the grammar of a PL. For example "Feature Modeling" notation [KKL$^+$98] can be used with bearing in mind the differences between patterns and features. Another example is the "Syntax Graph" that is used to show the syntax of programming languages, e.g., Pascal [Wir71]. Figure 7 shows the syntax diagram for the Request-Handling PL [BHS07b, p. 282].

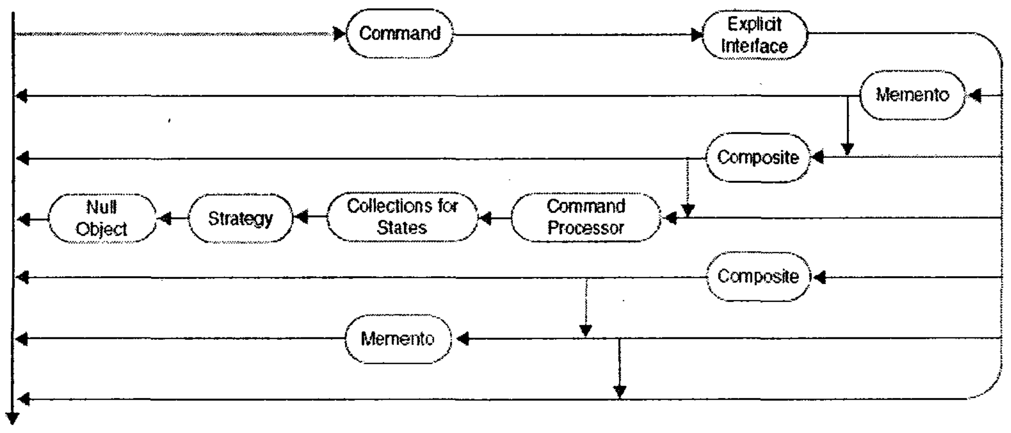


Figure 7: Syntax Diagram of the Request-Handling Pattern Language [BHS07b, p. 284]

More important than the notation chosen for the grammar, a PL must have clear guidance that shows the meaningful paths and prevents the designers from selecting ill-formed pattern sequences [BHS07b, p. 284].

The quality of a PL is reliant on both its maturity and its completeness. Maturity relates to the quality of the constituent patterns and their relationships. Completeness relates to the coverage of the problem and solution spaces by the language. Also the quality of a PL is related to the quality and maturity of its vocabulary (patterns) and its grammar (pattern dependencies) [BHS07b, p. 291].

Another Pattern Relationship Model Emphasizing on the fact that many of the researchers have ignored the importance of the relationship between patterns, Wo-dong et al. [WdKqY+03] have introduced a model for pattern relationships which is indeed a formalized and extended version of the "pattern graph" introduced by Alexander [A+77]. In a "pattern graph" there are two types of relationships between patterns: "Root" and "Leading." The authors have shown how the proposed model can be used in building frameworks.

The model has two parts: the pattern relationships, and the translating methods that convert the relationships into a component model. If Pattern Set (PS) be the set of all patterns in a PL, then the relationships between patterns is defined by the following definitions.

$ENTRY = \{< x >| \, x \in PS \wedge isarchitecturepattern(x)\}$

$LEAD = \{< x, y >| \, x \in PS \wedge y \in PS \wedge applied(x) \rightarrow toapply(y)\}$

$REQUIRE = \{< x, y >| \, x \in PS \wedge y \in PS \wedge applied(x) \rightarrow applied(y)\}$

$EXCLUDE = \{< x, y >| \, x \in PS \wedge y \in PS \wedge canapply(x) \rightarrow \neg \, canapply(y)\}$

$ALTERNATE = \{< x, y >| \, x \in PS \wedge y \in PS \wedge canapply(x) \leftrightarrow canapply(y)\}$

Based on this model, a Pattern Cluster (PC), a set which contains all the related patterns for a problem context, is defined as follows.

$PC \subseteq PS$

$\forall x, y \in PS : (ENTRY \; x \wedge x \; LEAD \; y) \Rightarrow y \in PC$

$\forall x \in PC \wedge y \in PS : x \; REQUIRE \; y \Rightarrow y \in PC$

$\forall x, y \in PC : \neg \, (x \; EXCLUDE \; y)$

$\forall x, y \in PC : x \; ALTERNATE \; y \Rightarrow x \; EXCLUDE \; y$

As it is clear from the above definitions, this method of framework development should start from an architectural pattern. Then, applying this root pattern leads us to other patterns that need to be applied. Maybe some of the applied patterns require other patterns to be applied. This process continues until the PC is completed. Meanwhile, the PC should remain consistent, meaning that conflicting patterns are not allowed to be added. The last two rules check the consistency of the under development PC.

2.4.6 Pattern Relationships in POSA-5

POSA-5 [BHS07b] is the last book in the Pattern-Oriented Software Architecture series which wraps up all the experiences and discussions of the previous volumes under the subtitle "On Patterns and Pattern Languages." We believe that it is one of the state-of-the-art references about PL and pattern relationships. However, PL field is still immature as the POSA-5 authors also emphasize that:

"not all the aspects of pattern languages we discuss in this part of the book are mature or well-established in the pattern community. For example, while fundamental aspects and properties of the process introduced by pattern languages, such as piecemeal growth, are widely accepted and practiced, other aspects and properties, such as the role of pattern sequences in defining a grammar for pattern languages, are considered as new or even subject to debate." [BHS07b, p. 245]

In this section, we present a brief review of the POSA-5 discussions by considering four type of relationships that could exist between the patterns: Competition, Completion, Combination, and Compound. We have also added more examples from different PLs.

Patterns in Competition

This relationship happens when there are more than one pattern to solve the same problem. The relationship can also be called pattern alternatives.

Following are some examples for patterns in competition. When the problem is "to fix the steps in an algorithm while allowing the implementation of the steps to vary," there are two GOF patterns available as solutions: STRATEGY and TEMPLATE METHOD [BHS07b, p. 138]. Two patterns of *PofEAA*, Optimistic Offline Lock [Fow02, p. 416] and Pessimistic Offline Lock [Fow02, p. 426], are alternatives for the problem of handling concurrency control issues. In POSA-4 [BHS07a], both OBJECTS FOR STATES and COLLECTIONS FOR STATES patterns "address the problem that an object's concrete behavior can depend on its current modal state" [BHS07b, p. 138].

When having several alternatives for a problem, the important challenge is "How to select one of the alternative patterns?" To answer, the key is to investigate "the context, the forces, and the consequences of competing patterns" [BHS07b, p. 144]. The *context* is one of the fields in the *pattern form*. The *forces* are the factors such as cost and performance. The *consequences* are the pros and cons of selecting each alternative. In addition to the above parameters, sometimes there are other subjective and cultural elements which affect our decision. Examples of these context information are as follows: programming language, complexity of the system, and expertise of the designer [BHS07b, p. 154].

While deciding on the competitive patterns during the design, we need to record the important discussions and investigations about the pros and cons that take place in selecting one of the alternatives. The resulted artifact is called "Design rationale" and is a useful document for future designers, and for maintainers of the system [PB88]

Patterns in Completion

This relationship exists when one pattern can structurally complement another pattern. This relationship can also be called patterns in cooperation. It can be considered as a stronger version of pattern usage or inclusion. For instance, consider the case that the COMMAND PROCESSOR contains the COMMAND pattern [BHS07b, p. 156]. Another example is the TABLE DATA GATEWAY pattern of *PofEAA* that needs RECORD SET as the return type of its find operations [Fow02, p. 144].

Patterns in Combination

There are cases that combining both alternative or cooperative patterns together results in better solution. This happens when we are not forced to apply an "exclusive-or" relationship between two patterns [BHS07b, p. 159]. For instance, the CLASS ADAPTER pattern can be nested within the scope of the OBJECT ADAPTER to obtain a solution that could not be addressed by either of the individual patterns.

Many of the *PofEAA* patterns can be combined together since the author's recommendations are not strictly forbidding the designer to combine the alternatives. For instance, in *PofEAA* Fowler [Fow02, p. 59] says "you can write the code in the style of either Transform View or Template View or in some interesting mix of the two."

Pattern Compounds

There exist patterns that one of their elements is also a pattern. Also we can group some patterns together to make a bigger pattern. Buschmann et al. [BHS07b, p. 166] defined a pattern compound as "a named, commonly recurring, cohesive combination of other patterns." As an example for the former, note that COMMAND pattern can be found inside the ENUMERATION METHOD, and for an example of the latter, consider combining COMMAND and COMPOSITE to obtain a COMPOSITE COMMAND pattern [BHS07b, p. 166].

In *PofEAA*, Front Controller [Fow02, p. 344] is an example of a pattern compound which has GOF Command pattern as its part. Figure 8 shows the structure of this pattern.

By scrutinizing some of the patterns that in the pattern community are known as pattern elements (atomic patterns), we will see them as pattern compounds. For instance, INTERPRETER was introduced first in GOF as a general-purpose pattern element. however, it can

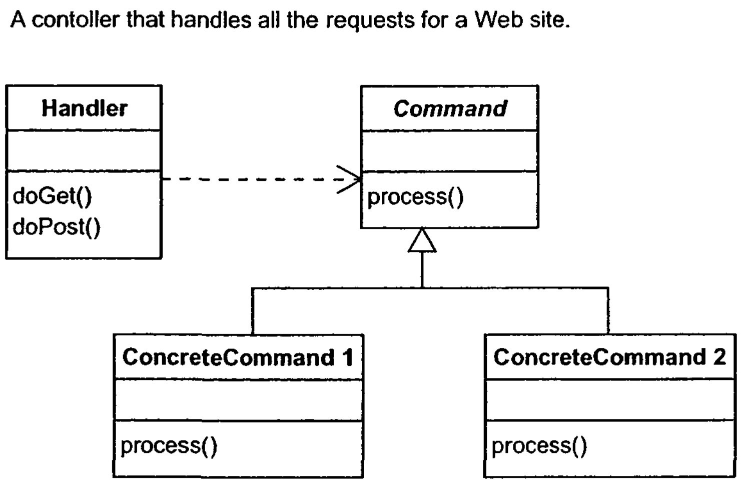


Figure 8: The Front Controller Pattern [Fow02, p. 344]

be interpreted as a pattern compound consisting of Command, Context Object, and Composite. Another example is the Model View Controller (MVC) pattern that in many works is considered as a single pattern, but a closer look at it reveals that it can be decomposed into either three elements (Model, View, and Controller), or seven elements (OBSERVER, COMMAND PROCESSOR, FACTORY METHOD, VIEW HANDLER, COMPOSITE, CHAIN OF RESPONSIBILITY, and BRIDGE) [BHS07b, p. 177-179].

Pattern Stories and Pattern Sequences

There are several ways for understanding "how a software system is designed" including investigation of its source code, diagramming its model, or recognizing its patterns. The results of all these methods are static, i.e., they show the system at a specific time. For example extracting the patterns used in a system is only a list: it does not show which pattern is used first [BHS07b, p. 184-185].

Storytelling is another method for describing the development of a system in a narrative way. A story tells us which pattern is used first, and what happened after. The story reveals the decisions made during the development of the system. One way of writing a story is to build a list of questions and answers. An example of a story is the presentation of the Lexi document editor in GOF [GHJV95, p. 33] which is used throughout the book to explain how patterns can be applied in practice [BHS07b, p. 185-189].

Following is a brief version of a story of the Request-Handling framework given in [BHS07b, p. 196].

COMMAND is expressed with EXPLICIT INTERFACE. COMMAND PROCESSOR is then introduced, to which COLLECTIONS FOR STATES is added. COMMAND is then augmented with MEMENTO. COMMAND PROCESSOR is then refined with STRATEGY, which leads to NULL OBJECT. COMPOSITE COMMAND is then introduced.

In a story there is no guidance on how patterns are connected together. Pattern sequences address this flaw by removing the story and talking on the order in applying patterns. It is worth noting that some PLs have not addressed pattern sequences at all; others made them implicit in the pattern descriptions. However, it is important to have pattern sequences as particular artifacts in a PL [BHS07b, p. 192-193].

By removing the story side from the above short story example, we reach to a more formal version as follows [BHS07b, p. 196].

¡COMMAND, EXPLICIT INTERFACE, COMMAND PROCESSOR, COLLECTIONS FOR STATES, MEMENTO, STRATEGY, NULL OBJECT, COMPOSITE COMMAND¿

Also some of the pattern compounds can be viewed as pattern sequences. For instance, decomposing the INTERPRETER pattern into its constituting patterns gives us the following tuple: ¡COMMAND, CONTEXT OBJECT, COMPOSITE¿ [BHS07b, p. 201]. There are several ways for presenting a pattern sequence. Maybe the simplest one is an ordered list of applied patterns [BHS07b, p. 193].

Buschmann et al. [BHS07b, p. 194-195] define a pattern sequence as "a successive progression of design decisions and transformations." They also emphasize that "a sequence represents a path through a design space [...] Following a pattern sequence is more like following a recipe than following a plan [...] How we choose between related pattern sequences will lead us to pattern languages."

Final note is that pattern context plays an important role in pattern sequences. It is the pattern context which tells us how to apply the pattern, as well as where in a sequence the pattern lies [BHS07b, p. 203].

Pattern Collections

Due to the fact that "Patterns are gregarious by nature" [BHS07b. p. 210], there is a tendency in presenting a set of patterns as a collection. The on-going work of Grady Booch's

Handbook of Software Architecture [Boo09], with about 2000 patterns, is the biggest collection of software patterns, ever. There are several approaches for organizing collections as described in the following [BHS07b, p. 211].

- Ad hoc organization: When there is no specific theme for organizing patterns. Pattern Languages of Program Design [MVN06] books belong to this group.

- Organization by level: When patterns are organized based upon the level of granularity. Three well-known packaging of patterns based on the level are: idioms, design patterns, and architectural patterns. As a simple definition, idioms are more fine-grained to be considered as a solution to a problem, i.e., an idiom is just a matter of convention. People use the term "design pattern" for a pattern which is similar to the GOF design patterns. Architecture patterns deal with most significant design decisions that shape a system [BHS07b, p. 213-216].

- Organization by domain: The domain can be divided into two parts: problem (application) domain and solution domain. The first deals with patterns that are related to the real world applications, such as health care or avionics. The second covers the software-centric concerns, such as architectural styles or programming languages. It should be noted that these two groups are not exclusive and designers need to consult patterns in both groups [BHS07b, p. 218].

- Organization by partition: When patterns are organized based on the part of the architecture in which they are applied [BHS07b, p. 219]. For instance, Fowler in *PofEAA* [Fow02] classifies his patterns into layers such as presentation, domain, and data source.

2.4.7 Pattern Relationships at a Glimpse

Table 3 summarizes this section by presenting the above discussion about the definitions and the formalisms of the pattern relationships. For each relationship, its name along with the references which have introduced that relationship are given. Further, another names for the relationship along with the references, the meaning of the relationship, some examples from Alexandrian patterns, GOF design patterns, *PofEAA*, and POSA-5 are presented.

Table 3: Pattern Relationships at a Glimpse

Relationship	*uses* [A+77, Zim95, GHJV95, Nob98a, NB02]
AKA	containment [A+77], completion [BHS07b], cooperation [BHS07b], requires [NB02], completes [NB02], follows [NB02]
Meaning	A *uses* B means pattern A uses pattern B in its solution, or B structurally complements A.
Alexander	Small Public Squares *uses* Pedestrian Density, Activity Pockets, and Something Roughly [Nob98a]
GOF	Observer *uses* Mediator, Mediator *uses* Singleton, MVC *uses* Observer, Strategy, and Composite [Nob98a], Interpreter *uses* Composite [NB02]
PofEAA	Front Controller *uses* Command, Table Data Gateway *uses* Record Set
POSA	Command Processor *uses* Command [BHS07b]
Relationship	*conflicts* [GHJV95, Nob98a]
AKA	competition [BHS07b], alternative [BHS07b, NB02], similar [Zim95]
Meaning	A *conflicts* B means patterns A and B are mutual exclusive solutions for the same problem
Alexander	House for a Small Family *conflicts* House for a Couple *conflicts* House for a Person
GOF	Decorator *conflicts* Strategy, Prototype *conflicts* Factory Method [Nob98a], Prototype *conflicts* Abstract Factory , Decorator *is alternative for* Strategy [NB02]
PofEAA	Optimistic Offline Lock *conflicts* Pessimistic Offline Lock
POSA	Objects for States *conflicts* Collections for States [BHS07b]
Relationship	*refines* [Nob98a]
AKA	specialization [NB02]
Meaning	A *refines* B means pattern B is a specialization of pattern A
Alexander	Sequence of Sitting Spaces *refines* Intimacy Gradient [Nob98a]
GOF	Factory Method *refines* Template Method [Nob98a], Factory Method *is a special kind of* Hook Method [NB02]
PofEAA	Data Mapper *refines* Mapper
Relationship	*combines* [Nob98a, BHS07b]
Meaning	A *combines* B means patterns A and B can be used together
Alexander	
GOF	Composite *combines* Iterator , Composite *combines* Visitor
PofEAA	Transform View *combines* Template View
POSA	OBJECTS FOR STATES *combines* COLLECTIONS FOR STATES
Relationship	*Compound* [BHS07b] **Notation:** $A \leftarrow B + C$
Meaning	1) Patterns B and C are joined together to make new pattern A 2) Pattern A can be decomposed into patterns B and C
GOF	1) COMPOSITE COMMAND ← COMPOSITE + COMMAND
POSA	2) MVC ← OBSERVER + COMMAND PROCESSOR + FACTORY METHOD + VIEW HANDLER + COMPOSITE + CHAIN OF RESPONSIBILITY + BRIDGE

2.5 Pattern Selection/Detection

2.5.1 GOF Design Pattern Detection

Most of the work on pattern detection is about detecting GOF design patterns. Some works try to find the patterns in the source code, others investigate models. Not all approaches are successful in detecting all GOF design patterns. There are patterns in GOF which are deterministically recognizable by checking the static models, e.g., Composite, and there are patterns that their structure is identical and their detection needs dynamic models (or even code) investigations, e.g., State and Strategy.

Tsantalis et al. [TCSH06] have proposed a design pattern detection which is based on similarity scoring between graph vertices. The idea is to represent each design pattern in term of a set of matrices, then, the given UML class diagram is also transformed to a set of matrices. The detection is performed by a tree search inside the given model to find an occurrence for a pattern. The approach is successful in detecting 20 of the 23 GOF design patterns.

Bergenti and Poggi [BP02] have developed a system called IDEA (Interactive DEsign Assistant) which detects design patterns in a UML diagram. The IDEA is integrated into both ArgoUML [Tig09a] and Rose [IBM09a]. The IDEA investigates both the class diagram and the collaboration diagram. The criteria that specify the structure of a pattern are written as Prolog rules. Eleven GOF design patterns are successfully detected by the IDEA.

Wuyts [Wuy98] has used a declarative reasoning approach (using Prolog rules) to describe the structure of GOF design patterns, and to detect the patterns in Smalltalk programs. For instance the following rule defines the structure of the Composite design pattern by using two other sub-rules.

```
head: compositePattern (?comp,?composite,?msg)
body: compositeStructure (?comp,?composite)
      compositeAggregation (?comp,?composite,?msg)
```

Kampffmeyer and Zschaler [KZ07] have built an ontology containing the *Intent* (see Section 2.3.3) of GOF design patterns. Then, they have built a tool that, given a problem, helps the designer choose the right pattern. All the 23 GOF patterns are covered in their work.

Blewitt et al. [BBS05] have introduced a prolog-like language named SPINE (see also [Tai07, chap. VI] and [Ble06]) as a pattern specification language. SPINE is used for defining a pattern in terms of constraints on its Java implementation. The authors have also shown how a proof engine named HEDGEHOG reads both the SPINE code and a Java code, and verifies the application of patterns in the Java code. From GOF design patterns, seven patterns could not be represented by SPINE.

Mak et al. [MCL04] have reused the idea presented by Guennec et al. [GSJ00] to present extensions for UML such that the recurrent structure and behavior of design patterns (pattern leitmotifs) can be specified precisely. The authors of both papers [MCL04, GSJ00] have concluded that current versions of UML (at the time of writing papers, 1.3 for[GSJ00] and 1.5 for [MCL04]) are ill-equipped for precise representing of design patterns. The work presented by the latter paper [MCL04] was believed to be a premier step in defining the UML 2.0 profile for the modeling of design patterns.

2.5.2 Pattern Enforcing Compiler (PEC)

Lovatt et al. [LSV05] (see also [Tai07, chap. XV] and [Lov06]) have built a system named Pattern Enforcing Compiler (PEC) to address the problem that different programmers implement a pattern in different ways. PEC is similar to a conventional Java compiler, with the extension of verifying the application of design patterns in the code. The interfaces are used as markers to inform PEC that a pattern is used. Hence, the patterns are enforced at the class level, not the instance level. The programmer has to use a 'boiler plate' code for the pattern that he/she wants to apply.

The PEC system is written for Java language and uses Javadoc to document pattern usages. It uses interfaces as markers for showing the developer's desire for applying a pattern. It is important to note that by selecting the interfaces as markers, the programmer knows beforehand which pattern he/she intends to use. PEC only shows 'pass' or 'fail' message to the developer. I.e., there is no advisory system. It uses a naming convention for easing the detection of class features. PEC generates most of the 'boiler plate' code for some of the patterns. PEC is extensible, meaning that the user can define new patterns without requiring any new syntax for the Java language.

Following are the criteria for a correct application of the Singleton pattern as indicated in the PEC Javadoc. A Singleton class has the following properties:

1. The class must be final.

2. The class must have a single, private, no argument, constructor that throws the exception `IllegalStateException` if it is called more than once.

3. If the class is serializable then it should have a `readResolve` method that returns the 'singleton.'

4. The class cannot be clonable.

5. The class must have a method called instance. This instance method must: always return the same object, have no arguments, be static, and have either package or public access.

The following is the 'boiler plate' code for the Singleton pattern given in [LSV05].

```
import pec.compile.singleton.*;
public final class SingletonClass implements Singleton {
  private final static SingletonClass instance = new SingletonClass();
  private SingletonClass() {
    if ( instance != null )
      throw new IllegalStateException("Attempt to create a second Singleton" );
  }
  public static SingletonClass instance() {
      return instance;
  }
  // other methods
}
```

The most important item in the above code is the line that tells the PEC that the `SingletonClass` implements the interface `Singleton`, i.e., this class is meant to be a Singleton. Then, PEC checks the criteria of the Singleton pattern. This is like the type checking mechanism of the Java compiler. For each pattern named X, there is a class named XUtility in the same package as the interface, that checks the structure of that pattern.

In addition to the static checking, PEC is claimed to have two more features: dynamic checking and code generation. Dynamic checking is used for the patterns such as Singleton that can not be detected only by static checking. For Singleton, an attempt is made to create two Singletons, if it is successful, an error is reported. Dynamic testing is done by the help of `java.lang.reflect` Application Programming Interface (API). As reported in the paper, seven patterns have been implemented in the PEC. However, new patterns can be added by a user which is familiar with Java, since PEC does not introduce any new syntax. Finally, the error messages given by the PEC are very simple, e.g., "Singleton classes must not be clonable."

2.5.3 Systematic Pattern Selection

Zdun [Zdu07] has proposed a systematic pattern selection approach which uses both PL grammars and design space analysis. An important prerequisite of this approach is to identify the relevant quality goals of the patterns. Quality goals can be found in the *forces* and *consequences* sections of the *pattern form*. After quality goals are found, the pattern relationships are formalized into a pattern language grammar, and the grammar is annotated with the effects of the selected patterns on the quality goals. A sample annotated pattern language grammar overview diagram is shown in Figure 9. The scores '++,' '+,' and '−' show the effect of the selected pattern on the specified quality goal. The [variants] mark should be interpreted as 'OR.' The diagram in Figure 9 can be converted to a formal grammar shown in Figure 10, using the notation presented in Section 2.4.5. Note that the capitalized words are the name of the patterns in the PL.

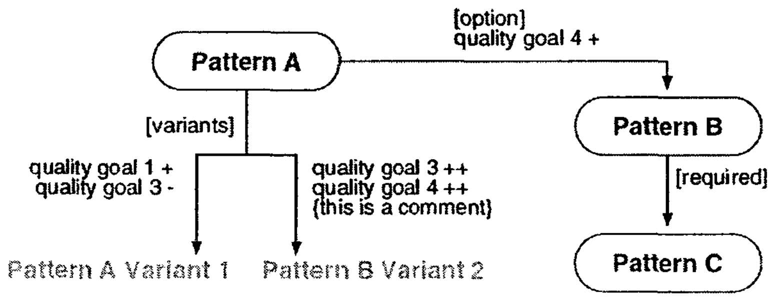


Figure 9: An Annotated Pattern Language Grammar Overview Diagram [Zdu07]

$S \rightarrow PatternAs\ PatternA\_Options$
$PatternAs \rightarrow PatternAs\ PatternA\ \mid\ PatternA$
$PatternA \rightarrow PATTERN\ A$
$PatternA \rightarrow PATTERN\ A\ VARIANT1$
$PatternA \rightarrow PATTERN\ A\ VARIANT2$
$PatternA\_Options \rightarrow \lambda\ \mid\ PATTERN\ B\ PATTERN\ C$

Figure 10: A Grammar Equivalent to the Diagram Given in Figure 9, Revised from [Zdu07]

As an example of a pattern sequence that can be derived from this grammar, consider the following derivation which results in applying only one pattern "*PATTERN A*."
$S \rightarrow PatternAs\ PatternA\_Options \rightarrow PatternAs \rightarrow PatternA \rightarrow PATTERN\ A$

The grammar helps the designer in several ways: in understanding the topology of the PL, in showing the possible pattern combinations, and in reviewing the effects on quality goals of the patterns. The grammar suffices for most (simple) design decisions, e.g., "if pattern A requires pattern B, the design decision is already clear." However, for complex design decisions, a design space analysis, using the Questions, Options, and Criteria (QOC) notation, is performed to reduce the complexity of the pattern selection process. A template for design space visualization using QOC technique is shown in Figure 11.

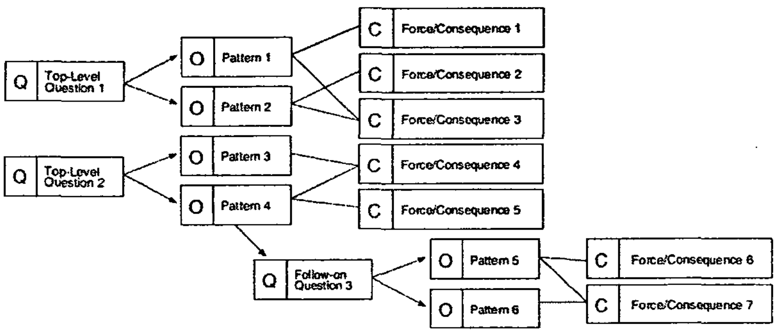


Figure 11: A Template for Design Space Visualization using QOC Approach [Zdu07]

This approach has been applied for a case study on "Remoting Patterns." The idea has also been validated via several academic and industrial projects, e.g., "re-engineering a document archiving system." As a track for future work it is claimed that "the pattern language grammars and design spaces can potentially be used as an input for model-driven tools" [Zdu07].

2.6 *PofEAA* PL

In this thesis, our focus is on Martin Fowler's book titled "Patterns of Enterprise Application Architecture" *PofEAA* [Fow02]. The book consists of a set of patterns for designing the architecture of a web-based enterprise application. Enterprise systems are more complex than other kinds of software considering the complicated business rules and the amount and complexity of data. These systems usually deal with huge amount of data (e.g., tens of millions of records) which needs to be persisted and accessed by many users concurrently, and to be integrated with other applications. Examples of enterprise applications are financial systems, reservation systems, and supply chain systems [Fow02, p. xviii].

While there is no consensus on the definition of the term "Architecture," most people agree that it is "the highest-level breakdown of a system into its parts" [Fow02, p. 1]. In software design, several approaches are introduced for the architectural design of a system, pipes, filters, and layers to name a few. Layering architecture is selected in *PofEAA*, that means, the book is about how to decompose a system into layers and how these layers work together [Fow02, p. 2]. Choosing an appropriate architecture for an enterprise application is hard. Based on the author's experience with enterprise applications, 51 patterns are introduced as solutions to the recurring problems that designers encounter while *designing* the architecture of a web-based enterprise application.

It should be noted that enterprise applications are not all the same, hence, there is no "one size fits all." That's why the author of *PofEAA* emphasizes that "many of the patterns are about choices and alternatives" [Fow02, p. 6]. Also it is clarified that patterns should not be used blindly and the designer needs to select a "half-baked" pattern and then modify it to meet his/her demands [Fow02, p. 10].

As it was mentioned earlier, in defining a pattern, each pattern author selects a *pattern form.* The pattern form used in *PofEAA* includes eight items: The pattern name, The intent, The sketch, A motivation problem, How it works, When to use it, The further reading, and The Examples [Fow02, p. 11].

In the following, first we briefly review how the *PofEAA* patterns are organized, then we discuss how this set of patterns is qualified to be considered as a PL.

2.6.1 Organization of the Patterns in *PofEAA*

Based on the idea of three-tiered architecture for object-oriented client-server platforms, the patterns in *PofEAA* are decomposed into three main layers [Fow02, p. 19]. Also there are supporting patterns for the issues such as object to relational conversion and concurrency management.

In the following, first we introduce the patterns of three main layers, then we see the supporting patterns. Note that grouping patterns into "main patterns" and "supporting patterns" is our choice and is not explicitly done in the book. Also note that the pattern names are in italic.

Main Patterns

Presentation Layer This layer is responsible for the user interface, i.e., displaying information and handling user requests. The patterns address the design problem "How does the system communicate with the user?" There are seven patterns in this layer: *Model View Controller*, *Page Controller*, *Front Controller*, *Template View*, *Transform View*, *Two-Step View*, and *Application Controller*.

Domain Layer This layer deals with application and domain logic and business rules, hence it is often called "business logic layer." The patterns address the design problem

"How the business logic of the system is organized?" There are four patterns in this layer: *Transaction Script, Domain Model, Table Module,* and *Service Layer.*

Data Source Layer This layer is related to communicating with the database, the messaging system, and other external applications. The patterns address the design problem "How does the system access the data source?" There are four patterns in this layer: *Table Data Gateway, Row Data Gateway, Active Record,* and *Data Mapper.*

A designer can find various answers for the mentioned design questions based upon a number of conditions, e.g., the platform he/she is working on, knowledge of developers, complexity of the domain objects, or complexity of business logic for each scenario. As it was mentioned earlier, the *PofEAA* does not force a single solution for a problem. Instead, several choices and alternatives are proposed.

For the **Presentation Layer**, using *Model View Controller (MVC)* is recommended. The word "Controller" is divided into two types by the author of *PofEAA*: "Input Controller" and "Application Controller." The Controller in MVC pattern is actually an Input Controller, but an *Application Controller* acts as "a separate layer that mediates between the presentation and domain layers" [Fow02, p. 58]. Only if the screen flow of the system is controlled by a machine, we need an *Application Controller.*

Two sets of patterns are introduced in this layer, one for *control* part and the other for *view* part. For the control part, two patterns are presented: *Front Controller* and *Page Controller.* Based upon the choice made for the domain layer, the technology we are using, and the complexity of the user requests, we may choose different patterns for the Presentation Layer. For example, *Front Controller* fits best with the Java technology and object-oriented modeling of domain concepts in Domain Layer (that means using *Domain Model* pattern), while *Page Controller* is simpler and could be used with a simple pattern for Domain Layer such as *Transactions Script.*

For the view part, three patterns are introduced: *Template View, Transform View,* and *Two-Step View.* Selecting either of the first two patterns, we have two options, to use it as a single stage, or a *Two-Step View.*

For the **Domain Layer**, there are three patterns: *Transaction Script. Domain Model,* and *Table Module.* in addition to an extra pattern, named *Service Layer*, which (optionally) can be used over *Domain Model* and *Table Module. Transaction Script* is used when the designer does not want to model this layer with object-oriented technology. It is just a bunch of code for each transaction of the system put in a method or a separate class. When developers are not familiar with object-oriented technology, this choice is probably the best one for the business logic layer. On the other hand, *Domain Model* is based upon object-oriented design of domain concepts. This is the most appropriate choice for object-oriented developers using technologies such as Java and C++. *Table Module* works best for people working with .Net or Visual Studio which have facilities for things like *Record Set* which is a base pattern. As Fowler says, "These three patterns are not mutually exclusive choices. Indeed, it's quite common to select *Transaction Script* for some of the domain logic and apply *Table Module* or *Domain Model* for the rest" [Fow02, p. 30].

Table 4 summarizes the discussions given in the *PofEAA* book in terms of advantages and disadvantages of each pattern from the Domain Layer. According to the table, one of the most important factors in choosing the right pattern for structuring the domain, is the complexity of the domain logic. Unfortunately, there is no metric for measuring this complexity. One solution is to ask an expert to review the requirements and give you a judgment [Fow02, p. 30].

Table 4: Alternative Patterns for Domain Layer (Adapted from [Fow02, p. 25-32]).

Pattern	**Advantages**	**Disadvantages**
Transaction Script	Easy to use and understand for most developers. Easy to build atop a relational DB. A simple procedural models.	Does not fit with the complex business logics. Duplicate code is inevitable.
Table Module	Works well with moderate business logic. Easy for connecting to relational DB.	Does not fit with the complex business logics.
Domain Model	Handles complex business logic in a well-organized way. Matches well with OO paradigm.	Hard to use and understand for non-OO people. Difficult for connecting to relational DB. Object/Relational mappings are needed.

In the **Data Source Layer**, depending on the choice made for Domain Layer and the complexity of the domain objects, along with the properties of the database, designer will select the appropriate pattern for talking to the database. Sometimes there is a simple mapping between the objects and the database tables, but this mapping is not always straightforward. There are two patterns named *Mapper* and *Gateway* to solve this problem. *Mapper* is used when there is a cross table transaction to access the information needed to construct an object, but *Gateway* simply packs the queries for one table and is used when a domain object can fulfill its assigned tasks just by accessing one table.

There are two kinds of Gateways: *Table Data Gateway* and *Row Data Gateway*. One instance of the former handles all the rows in the table which usually contains a mini-table such as *Record Set*, but for the latter, there is one instance per each row of a table.

Sometimes a user request can be responded by just one domain object talking to just one database table. In these cases, the logic in the domain object and the queries in the gateway can be combined in an object called *Active Record* which is another pattern of the Data Source Layer [Fow02, p. 35].

Supporting Patterns

Object-Relational patterns These patterns are applied when the relational database is used for storing the objects. There are 16 patterns in this group which are divided into three subgroups named Structural, Behavioral, and Metadata Mapping. The first deals with the problem of relating objects to tables, the second concerns about loading and saving objects to the database, and the third is related to the actual task of object-relational mapping. *Single Table Inheritance*, and *Class Table Inheritance* are example patterns in the first group; *Unit of Work* and *Identity Map* are examples in the second group; *Metadata Mapping* and *Query Object* are examples in the third group.

Distribution patterns These patterns address the issues of distributing an application on different nodes. Due to the pitfalls of distributed architecture, the first recommendation given by the author of *PofEAA* encourages the designers not to distribute their objects! Then to limit the distribution boundaries as much as possible. Finally, there are two simple patterns named *Remote Facade* and *Data Transfer Object* in this group that can help the designer manage the distribution of objects.

Offline Concurrency patterns These patterns are simple techniques that help the designer deal with concurrency control issues, although they are merely starting points and there is no guarantee that they can cure all concurrency problems. Concurrency problems occur when there are multiple processes or threads manipulating the same data. However, dealing with concurrency is one of the hardest issues in software development.

As a naive solution, it is recommended to do all the data manipulations within a transaction. Even with this solution, there still exist some concurrency issues, which in the book are referred as *offline concurrency*, "that is, concurrency control for data that's manipulated during multiple database transactions" [Fow02, p. 63]. There are four patterns in this group: *Optimistic Offline Lock*, *Pessimistic Offline Lock*, *Coarse Grained Lock*, and *Implicit Lock*. Table 5 shows the pros and cons of the alternative patterns for managing concurrency in a system.

Table 5: Alternative Patterns for Concurrency Control (Adapted from [Fow02, p. 68,77]).

Pattern	Advantages	Disadvantages
Optimistic Offline Lock	Easy to implement. Better Concurrency	Needs to redo the task in case of conflict. Late discovery of fail.
Pessimistic Offline Lock	Early discovery of fail.	Hard to implement. Reduces concurrency.
Coarse-Grained Lock	Manages the concurrency of a group of objects together.	
Implicit Lock	Saves time of managing locks directly. Avoids bugs.	

Session State patterns The patterns in this group address the issues of having stateless versus stateful sessions. The first suggestion is to have everything stateless! But for the cases that you ought to have stateful sessions, there are three patterns presented in this category: *Client Session State, Server Session State,* and *Database Session State.*

Base patterns This group contains 11 patterns which are more general and localized and will be referred in the discussion of other patterns. Examples are *Gateway, Mapper*, and *Record Set.*

2.6.2 *PofEAA* as a PL

Despite the fact that the author of *PofEAA* does not force the designers to select one pattern after another, and says "I've tried to make each pattern as self-standing as I can" [Fow02, p. 10]. However, there exist several prominent features in this set of patterns that enable us to argue that *PofEAA* is a PL for the *design* of web-based enterprise applications. The facts that support this idea are as follows.

1. Fowler says "the value [of patterns] lies in helping you communicate your idea [...] The result is that patterns create a vocabulary about design, which is why naming is such an important issue" [Fow02, p. 11]

2. *PofEAA* patterns are closely related to each other and can be used to design the architecture of a web-based enterprise application.

3. There are several recommendations and great suggestions in *PofEAA* about how to decide amongst various alternative patterns for the situations that the designer has to select one pattern amongst the available choices. For instance, it is said: "A simple Domain Model can use Active Record, whereas a rich Domain Model requires Data

Howdo I structure my domain logic?

The logic is simple → *Transaction Script (110)*

The logic is complex → *Domain Model (116)*

The logic is moderate and there are good tools around Record Set (508) → *Table Module (125)*

How do I structure a Web presentation?

→ *Model View Controller (330)*

How do I interact with the database?

I'm using Transaction Script (110) → *Row Data Gateway (152)*

I'm using the Transaction Script (110) and my platform has good support for Record Set (508) → *Table Data Gateway (144)*

I have a Domain Model (116) that corresponds closely to my database tables → *Active Record (160)*

I have a rich Domain Model (116) → *Data Mapper (165)*

I'm using Table Module (125) → *Table Data Gateway (144)*

Figure 12: A Cheat Sheet for Selecting Patterns [Fow02, inside back cover]

Mapper" [Fow02, p. 117]. This set of recommendations and suggestions, can be considered a structuring mechanism that leads a novice designer in selecting appropriate patterns one after another and hence design a system.

4. The set of patterns in *PofEAA* is rich enough to describe the design of an application as a whole. That means we can find the problem along with its solution in *PofEAA* for almost any enterprise need and these patterns guide us through the design of each part of the application and consequently the whole system.

As an informal version of the structuring mechanism, Fowler has augmented the *PofEAA* book [Fow02, Inside Back Cover] by a "Cheat Sheet." In Figure 12, we have selected an excerpt from the Cheat Sheet that aims to guide the designer in selecting appropriate patterns for the Domain Layer and the Data Source Layer.

As a typical road map for building the architecture of a web-based enterprise application, a developer starts from the Domain Layer and based on the factors such as the complexity of the domain logic, the difficulty of connecting to a database, and the using tools, selects one of the three contender patterns.

After the decision about the Domain Layer is made, the developer goes down to the Data Source Layer and thinks about how to connect the domain objects to the data sources. At this stage, the decisions are dependent upon the Domain Layer choice along with extra factors such as the facilities provided by the implementation platform and the complexity of the domain model.

Final step would be selecting patterns for the Presentation Layer, although the presentation is not tightly dependent upon the choice of the lower layers. There are two main options for this layer: a rich-client interface or an HTML browser interface. The choices have to be selected amongst the latter since no pattern is presented for the former. The recommended pattern is the *Model View Controller* which leads the designer to decide about the controller and the view. These two last decisions will be affected by the development tool.

In the course of designing a system, designer should remember that there is always possibility of making mistakes or sometimes it is required to improve the design. As Fowler says "Architectural refactoring is hard [...] but it isn't impossible," [Fow02, p. 95] that means, while the designer should be prudent in designing a system, he/she should not resist change when it is required.

2.6.3 *PofEAA* is in the Solution Domain

The word domain in software engineering means "an application area or field for which software systems are developed" [Rub90]. When we talk about the *problem domain*, we mean the concepts which are related to the corresponding business problem of the application. However, the *solution domain* deals with the concepts related to the implementation details of the system.

It is obvious that *PofEAA* is a PL for *designing* web-based enterprise applications and not a language for the analysis of web applications. Therefore, *PofEAA* should be considered as a PL in the *solution domain* describing the architecture or *design* of the application.

Chapter 3

Pattern Language Verifier (PLV)

In Section 2.3.4, we accepted the following definition for a Pattern Language (PL):

> "A network of tightly interwoven patterns that defines a process for systematically resolving a set of related and interdependent software development problems" [BHS07b, p. 260].

This definition emphasizes that in building software based upon the patterns of a particular PL, i.e., in *designing with patterns*, the application of the patterns is not arbitrary, i.e., the designer must adhere to the relationships between patterns.

Due to the fact that a PL may contain dozens of patterns with a variety of possible relationships between them, and the relationships are mainly embedded into the lengthy texts of pattern descriptions, *designing with patterns* is not an easy task, particularly for a lay person. For instance, Patterns of Enterprise Application Architecture (*PofEAA*) [Fow02] consists of 51 patterns with relationships such as uses, alternative, and conflicts, which are all explained in prose description.

Following are some of the challenges that the designers face when they want to utilize the patterns of a PL in designing a model for their software.

1. Pattern Selection: Which pattern is appropriate to solve a particular problem?
2. Pattern Application: How to apply a pattern correctly?
3. Pattern Weaving: Which pattern to choose after a specific pattern?
4. Pattern Semantics: Which pattern matches best with my Data Base Management System (DBMS) or my implementation language?

In this chapter, we propose a process named Pattern Language Verifier (PLV) to assist designers in verifying the application of a PL in the design. Because of the nature of PLV, which is a verifying process, we exclude the "Pattern Selection" challenge from the PLV's duties. That means, PLV aims to help a designer when facing the items two, three, and four of the above list. We suppose that, the designers know which pattern they want to use, and they show their intention by using the name of the pattern. In Chapter 2, we introduced works that help designer in pattern selection.

For every PL, if the PLV is defined, it helps a designer find answers to the questions such as the following.

1. Is the usage of pattern X in my design correct?

2. Is it correct to use pattern Y as an alternative for pattern X?

3. Is my model consistent considering my implementation tool?

To the best of our knowledge, PLV is the first work which addresses the problem of verifying a design model from the PL view. Most of the related work cited in Chapter 2 falls into the category of single pattern detection and those works do not focus on the PL aspects.

The PLV process is inspired by the compiler idea. Compilers look at the tokens of a programming language as words, and the programs as sentences. They say a programming language must have rules that define the legal words, the correct format of the sentences, and the meaningfulness of the sentences.

We believe that checking a model which is built using the patterns of a PL is similar to using a compiler for checking a source program which is written in a programming language. This similarity is the cornerstone of defining the PLV process. The idea of similarity between PL and a formal grammar is also pointed out by other researchers [NB02, HAZ07, Zdu07, BHS07a].

Buschmann et al. [BHS07a, p. 13] say "While patterns represent a design vocabulary, pattern languages are somewhat like grammar and style." Harrison et al. [HAZ07] mention that as an ongoing research "We propose deriving a pattern language's grammar to systematically describe the pattern relationships and annotating the grammar with effects on quality goals." Zdun [Zdu07] uses annotated PL grammars in systematic pattern selection. Noble and Beedle [NB02] argue that the PL concept proposed by Alexander [A^+77, Ale79] represents "a tree or directed graph of patterns, similar in structure to a formal grammar." We have individual patterns, one of them is considered as *initial pattern*. Then via *uses* relationship, like a production rule in a grammar, we reach to other patterns to apply. The *initial pattern* is like a grammar's start symbol, and addresses a large scale problem.

The remainder of this chapter is organized as follows. Section 3.1 reviews the compilation process. In Section 3.2, we discuss the overview of the PLV process. Section 3.3 elaborates on the rules as an important prerequisite for PLV. Section 3.4 discusses the similarities and differences between the PLV process and the compiling process. Section 3.5 shows a version of PLV as a profile-driven process, and presents the technical details about its modules. In Section 3.6 we discuss the issues and points of improvement regarding the PLV process.

3.1 The Compilation Process

A compiler checks a program source code and generates the machine code. In compiler design, it is recommended to break the compiling task into two parts: an *analysis part* and a *synthesis part* [ASU86]. The former part mainly consists of three *phases*, Lexical Analysis (aka Scanning), Syntactic Analysis (aka Parsing), and Semantic Analysis. One of the responsibilities of those phases is to detect the lexical, syntactic, and semantic errors in a program respectively. The latter part, which is not related to our work and therefore is not discussed here, deals with the code generation and code optimization.

The Lexical Analyzer accepts the program source code as input and produces a list of tokens to be used by the next phase. This phase uses the *lexical rules* of the underlying language as a reference for detecting tokens. A token is a sequence of consecutive characters from the source code that together makes a meaningful logical unit (or word). The Lexical Analyzer is concerned only with recognizing tokens, i.e., it does not concern with the order of tokens. Tokens are typically divided into groups, identifiers, keywords, and operators, just to name a few.

The Syntactic Analyzer accepts a stream of tokens, generated by the Lexical Analyzer, and builds an abstract syntax tree (parse tree) based upon the given stream of tokens. This phase uses the formal specification of the source language which is mainly in the form of a *grammar*. A grammar consists of a set of rules that determine whether or not a sequence of tokens builds a correct sentence of a language.

The duty of Semantic Analyzer is to check whether or not the given parse tree is meaningful. As the reference for detecting semantic errors (i.e., determining the meaningfulness of a program), the Semantic Analyzer uses the *semantic rules* of the source language. Since formal specification of the semantics of a programming language is not easy, one alternative approach which is widely used by compilers is to augment the grammars with semantic rules. The resulting grammar is called an *augmented grammar.* The Semantic Analyzer augments the given parse tree with attributes and generates an annotated parse tree. This phase checks the semantic features that cannot be specified by the grammar.

The above three phases make use of a Symbol Table as a common source for accessing the information about variables and identifiers in a program. Also there is a common unit called Error Handler which is responsible for reporting errors and performing error recovery tasks [ASU86].

3.2 The PLV Process

Following the three-phase *analysis part* of a compiler, we propose three main verification phases (modules) for the PLV process as indicated in Figure 13.

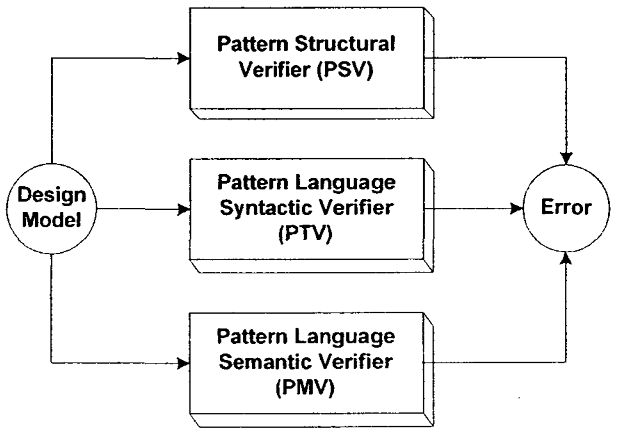

Pattern Structural Verifier (PSV)
Design Model
Pattern Language Syntactic Verifier (PTV)
Error
Pattern Language Semantic Verifier (PMV)

Figure 13: Three Phases of the PLV Process

1. Input to the process is a **Design Model**. By a Design Model, we mean a set of UML diagrams which shows the architecture and/or the detailed design of a system. We assume that the Design Model is built based upon the patterns of the underlying PL. The Design Model may contain any of the UML diagrams, but PLV only investigates the package diagrams and the class diagrams. From now on, we may use the word "model" to refer to the Design Model.

2. **Pattern Structural Verifier (PSV)**: PSV reads the Design Model, and applies the *structural* rules of the PL to verify that all the individual patterns that are used by the designer, are structurally correct. Our assumption is that the designer explicitly

shows his/her intention in using a pattern by applying the name of the pattern. For each structural error, PSV gives an error message to the designer.

3. **Pattern Language Syntactic Verifier (PTV)**: PTV accepts the Design Model and uses the *syntactic* rules of the PL to verify that the pattern combination detected in the previous phase is syntactically correct. The syntactic checks include verifying the organization of patterns and the relationships between used patterns. For each error, an error message is given to the designer.

4. **Pattern Language Semantic Verifier (PMV)**: PMV accepts the Design Model, and applies the *semantic* rulesof the PL to verify that all the patterns detected by the PSV and their relationships are semantically correct. Consistency between the parameters of methods, consistency between the patterns and their layers, and consistency between the patterns and the context information are considered as semantic rules in this work. For each error, an error message is given to the designer.

5. Output of the process are **Error** messages shown to the designer informing him/her of the problems in the design.

3.3 Rules: Important Requirement for PLV

In the previous section, we have repeatedly mentioned that each PLV module applies corresponding rules to check the model. In the heart of PLV there exist three classes of rules: structural , syntactic, and semantic. But, "what are these rules?" and "how should we prepare these rules and feed them into PLV?" In the following, we will elaborate on the nature of these rules and the items in the *pattern form* that help us recognize them.

3.3.1 Structural Rules

The structural rules are the basis used by the PSV phase in order to verify the structure of each individual pattern that is applied in the design model. By structural rules of a PL, we mean sets of rules where each set shows the essence of one pattern in the language. The structural rules of a pattern must address the following questions [GHJV95, p. 3]:

- What are the constituting elements of the pattern?
- How the elements are connected to each other to form the whole pattern?
- What are the responsibilities of the elements?

- How do the elements collaborate with each other?

Since patterns act as the words that the sentences of a PL are built based upon, it is crucial that, every single pattern is precisely defined by structural rules. It is obvious that having more accurate rules results in a more precise PSV. In compilers, the lexical rules that define the words of a language, must deterministically decide whether or not a token belongs to that language. To have determinism in detecting a pattern by PSV, we need precise structural rules.

Where to find structural rules?

We want to know which items in a *pattern form* must be investigated to find the structural rules of a pattern. Recall the classical definition for pattern that says "a pattern is a solution to a problem in a context" [GHJV95, p. 3]. Hence, the structure of a pattern is shown via the items of *pattern form* that represent the "solution" proposed by the pattern. Since there is not a unique or standard *pattern form* for writing patterns [Fow03], finding the *pattern form* items that show the structure of a pattern is not straightforward. We must start from the items such as "Structure" or "Solution," however, if these fields do not suffice, it is required to investigate other fields of the *pattern form* to better understand the structure of a pattern.

For patterns that their structure is represented by class diagrams, e.g., GOF patterns, the structural rules are the interpretation of these class diagrams. In GOF *pattern form* there is a field named "Structure" that shows the structure of the pattern by class diagrams. In some cases, e.g., Flyweight pattern, the structure contains an object diagram too. The class diagrams given in "Structure" fields of GOF patterns are rich enough to be considered as the structural rules of this PL. However, more information about the essence of the GOF patterns can be gained via field "Participants" which shows the elements of the pattern, or via field "Implementation" which shows how the pattern can be implemented in a programming language such as C++.

In *PofEAA*, for some of the patterns there is a class diagram that shows its structure: Front Controller, Remote Facade, to name a few. Other patterns, e.g., Layer Supertype, lack any diagram and are explained via textual description and example. We found the following fields of the *PofEAA pattern form* useful in finding the structural rules of a pattern. The list is sorted on the importance of the items.

1. "Intent," which is a short text about the pattern;

2. "Sketch," which is a visual representation of the pattern, mostly in terms of a UML class diagram;

3. "How It Works," which is a text for describing the solution; and

4. "Examples," which are Java or C# source code.

Formalism for structural rules

There are some design/architectural PLs, e.g., GOF [GHJV95], that use UML (mostly class diagram) as a formalism for representing the structure of patterns. Therefore, it is tempting for us to select UML as the language for representing the structural rules of the PL. However, investigating the patterns of these languages reveals that, in many cases, the pattern authors had to augment their UML diagrams by additional prose explanations to enrich the definition of the patterns or to shed light on the vague points. There are PLs, e.g., *PofEAA* [Fow02], that use UML diagrams for some of their patterns (not all of them). Some PLs, e.g., POSA-4 [BHS07a], use visualizations other than UML for defining the patterns, and their focus is on textual description.

Therefore, the formalism that we use for defining the structural rules is a combination of textural rules that are written in plain English with a UML class diagram. The textual rules, written as an enumerated list, are the main parts and must be clear and simple enough such that an intermediate OO programmer can understand them and interpret them in terms of an OO programming language such as Java. The class diagram is the complementary part that is optionally added to help understanding of the textual rules.

To facilitate the detection of elements in a model, and hence, to ease the work of PSV, we utilize a naming convention paradigm. We suppose that, the designers, in their design models, use the same names that are used by the pattern author in the *pattern form* for naming (or building the name of) the pattern elements. From now on, we refer to the name of the pattern as "Sign" [ZKB08]. For instance, if the designer intends to use the Table Data Gateway pattern [Fow02, p. 144] in accessing the Person information, he/she may choose "PersonTableDataGateway" as the name of the class which corresponds to this pattern.

Examples of structural rules

In order to see how the structural rules for a specific pattern can be extracted, we consider two different representations (see Figure 14 and Figure 16) for the Table Data Gateway pattern.

Figure 14 shows the "Intent" and the "Sketch" of this pattern in *PofEAA* [Fow02, p. 144]. Although, the intent text and the sketch give much information about the structure of the pattern, however, investigating other fields such as "How It Works" adds more details about the pattern. For example, we found the following comments very beneficial:

- "A Table Data Gateway has a simple interface, usually consisting of several find methods to get data from the database and update, insert, and delete methods" [Fow02, p. 144].
- "The trickiest thing about a Table Data Gateway is how it returns information from a query. Even a simple find-by-ID query will return multiple data items" [Fow02, p. 144].
- "One alternative is to return some simple data structure, such as a map. [...] A better alternative is to use a Data Transfer Object. [...] To save all this you can return the Record Set that comes from the SQL query" [Fow02, p. 144-145].

The first comment reveals that there could be more than one "find" method in this pattern. The second comment leads us to a rule that checks the return type of *all* "find" methods in this pattern. The third comment tells about the alternative options for the return type of "find" methods, e.g., to expect a Record Set as their return type. Note that, if the designer is going to define and use the Record Set as a pattern in his/her design, this rule will be considered as a syntactic rule (see Section 3.3.2).

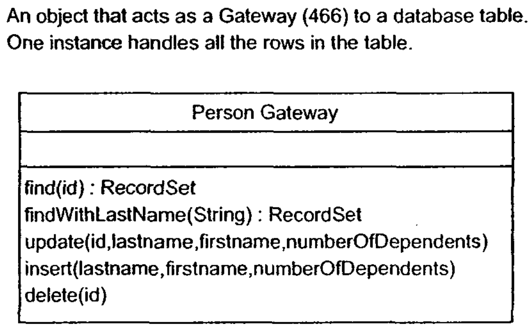

An object that acts as a Gateway (466) to a database table.
One instance handles all the rows in the table.
Person Gateway
find(id) : RecordSet
findWithLastName(String) : RecordSet
update(id,lastname,firstname,numberOfDependents)
insert(lastname,firstname,numberOfDependents)
delete(id)

Figure 14: The Table Data Gateway Pattern [Fow02, p. 144]

Considering the above discussion of the Table Data Gateway pattern, we are now able to define the structural rules for correct application of this pattern. Figure 15 shows these rules. Note that, the name "Table Data Gateway" is the "Sign" of the pattern, e.g., it can be used as part of the class name. Adding the class diagram in Figure 14 to these rules is not necessary since the rules are clear enough.

1. There is a `Table Data Gateway` class in the model.
2. There are at least four operations (`find, insert, delete, update`) in the `Table Data Gateway` class.
3. The return type of all `find` operations is `Record Set`.

Figure 15: Structural Rules for Table Data Gateway Pattern

Figure 16 shows the Table Data Gateway pattern presented in POSA-4 [BHS07a, p. 544]. In the *pattern form* adapted by this PL, there are several items for presenting the "Solution" offered by a pattern [BHS07a, p. 48]. In the figure, you see the "Solution Instruction" and "Solution Sketch" items for the Table Data Gateway pattern.

Wrap the database access code for a specific database table within a specialized table data gateway, and provide it with an interface that allows applications to work on domain-specific data collections.

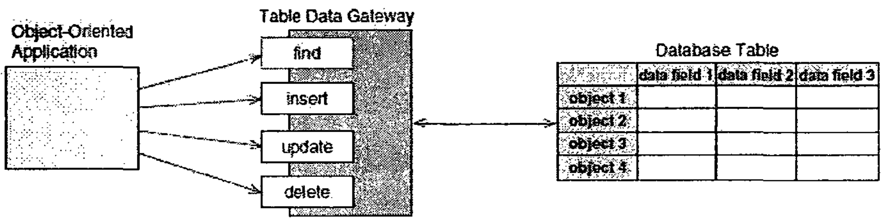


Figure 16: The Table Data Gateway Pattern [BHS07a, p. 544]

As it is clear from the figure, the formalism used for the sketch of the solution is not UML. Therefore, in addition to the items shown in the figure, other fields of the *pattern form* must be investigated to interpret clear structural rules that enable PSV to verify whether or not part of a design matches the Table Data Gateway pattern.

For instance, the following two sentences give us more information about the structure of the pattern.

- "A table data gateway has a simple interface consisting of several find methods to get data from the database, together with corresponding update, insert, and delete methods" [BHS07a, p. 545].
- "Many alternatives exist for returning the results of queries to clients [...] Some environments [...] can return a RECORD SET" [BHS07a, p. 545].

By considering this information, we reach the same structural rules that were extracted from the *PofEAA* representation of Table Data Gateway pattern (Figure 15).

To summarize, the structural rules of a PL are the rules that show the structure of each single pattern in the language clearly and precisely. These rules must be written in plain English text, in itemized format, optionally enriched by a class diagram, such that an intermediate OO programmer is able to understand and code them into the PSV module.

3.3.2 Syntactic Rules

The syntactic rules are used by the PTV phase in order to verify the pattern combination that is applied in the design model. We believe that syntactic rules must address two aspects about the pattern combinations that are used in a design model: the organization of patterns, and the relationship between patterns. These two aspects are discussed in the following.

I- Syntactic rules regarding the organization of patterns

Patterns are normally organized into groups or layers. The syntactic rules of a PL must enforce the correct organization for the patterns that are applied in a design model.

The concept of pattern grouping is addressed by many pattern authors [A^{+}77, GHJV95, SSRB00, Fow02, KJ04, BHS07a]. For instance, the authors of GOF say, "Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them" [GHJV95, p. 9]. The GOF patterns are classified into three groups based upon their purpose: creational, structural, and behavioral [GHJV95, p. 10].

The concept of pattern layering becomes more tangible for architectural patterns, because, it is common that these patterns are divided into groups based upon the architectural style selected by the author. For instance, Fowler [Fow02, p. 19] has used the layered architecture for *PofEAA*, by dividing patterns into three primary layers and five supporting layers (see Section 2.6). The primary layers are in fact the mandatory layers that each enterprise application must include: Presentation, Domain, and Data Source. The use of supporting layers depends upon the designer's choice, the features of application, and the configuration of the system. Hence, in a design which is built using *PofEAA* patterns, missing any of the mandatory layers in a model, or placing a pattern into a wrong layer, should be considered as a syntactic error.

Where to find syntactic rules regarding the organization of patterns?

If the patterns of a PL are organized into groups or layers, this fact should be clearly distinguishable either from the *pattern form* or from the method of documentation of the patterns.

The first means that there is a dedicated field in the *pattern form* that indicates the corresponding group for each pattern. This approach is applied in GOF patterns, where a field named "Classification" is attached to the "Pattern Name" to make the field "Pattern Name and Classification" in the *pattern form*. "Classification" indicates the group that the pattern belongs to. For instance, "Command" is considered as a behavioral pattern [GHJV95, p. 6-10].

The second means that in documenting the patterns, the author places all the patterns that are in the same group under one title. This is the approach taken by *PofEAA* [Fow02] and POSA-4 [BHS07a]. In *PofEAA* book, patterns of the same layer are defined in the same chapter. Table 6 shows the organization of *PofEAA* patterns into layers along with the book chapter that the patterns are described in it. In POSA-4 book, there is a chapter dedicated to the patterns of each problem area.

Table 6: Organization of Patterns in the *PofEAA* Book, Adapted from [Fow02]

Layer/ Category	**Book Chap.**	**Patterns**
Domain Logic	9	Transaction Script, Domain Model, Table Module, Service Layer
Data Source	10	Table Data Gateway, Row Data Gateway, Active Record, Data Mapper
Object-Relational Behavioral	11	Unit of Work, Identity Map, Lazy Load
Object-Relational Structural	12	Identity Field, Foreign Key Mapping, Association Table Mapping, Dependent Mapping, Embedded Value, Serialized LOB, Single Table Inheritance, Class Table Inheritance, Concrete Table Inheritance, Inheritance Mappers
Object-Relational Metadata Mapping	13	Metadata Mapping, Query Object, Repository
Web Presentation	14	Model View Controller, Page Controller, Front Controller, Template View, Transform View, Two-Step View, Application Controller
Distribution	15	Remote Facade, Data Transfer Object
Offline Concurrency	16	Optimistic Offline Lock, Pessimistic Offline Lock, Coarse Grained Lock, Implicit Lock
Session State	17	Client Session State, Server Session State, Database Session State
Base	18	Gateway, Mapper, Layer Supertype, Separated Interface, Registry, Value Object, Money, Special Case, Plugin, Service Stub, Record Set

Although indicating the placement of patterns in layers is easy, e.g., by a two-column "layer/pattern" table similar to Table 6, such a table does not show all the details about the organization of patterns. For instance, the fact that three of the layers of *PofEAA* are mandatory and five layers are optional is not reflected in Table 6. Capturing this information needs scrutinizing of the pattern text. For example, the following excerpts from the *PofEAA* book, helps us to find out that three of the layers are principal, but, for example, the Distribution layer is optional.

- "For this book I'm centering my discussion around an architecture of three primary layers: presentation, domain, and data source" [Fow02, p. 19]. Fowler clarifies that these three layers are primary.
- "Hence, we get to my First Law of Distributed Object Design: Don't distribute your objects!" [Fow02, p. 89]. Fowler recommends not to distribute the objects, unless you have to do so.

Formalism for syntactic rules regarding the organization of patterns

How to formalize the organization of patterns of a PL for the PLV process? The formalism should precisely address questions such as the following. What layers exist in the language? Is there any order or dependency between layers? Which pattern lies in which layer? If a layer is optional, what are the prerequisites that must be true to have that layer in the model?

Following context-free grammars, BNF, and set notations, we define a notation in Table 7 for representing the organization of patterns into groups or layers. In terms of UML, since a "package" is often used to group elements [Lar05, p. 201], we correspond one layer/group to one package in the model. Hence, the syntactic rule that checks the membership of a pattern in a layer/group ($l \ni P$), should simply check that the main class of the pattern is placed in the corresponding package.

Note that in grammar terminology, lower case words are non-terminal symbols and are equivalent to the layers, capitalized words are terminal symbols and are equivalent to the patterns. The starting non-terminal of the grammar indicates the system as a whole.

Examples of syntactic rules regarding the organization of patterns

Using the organization of the *PofEAA* book along with the context of patterns that are explained in prose in the book, we can extract the syntactic rules regarding the organization of patterns. For instance, to enforce the fact that every design model which is built using *PofEAA* requires a root package, and inside that, there must be a "main" package and an optional "auxiliary" package, we write the rule: $pofeaa\ model \supset main\ layer\ .\ auxiliary\ layer^*$.

Table 7: Notations for Representing the Organization of Patterns

Notation		Meaning	In terms of UML
l m	Layer	Lowercase letters or first-small words show the layers	A layer is represented by a package
P Q	Pattern	Capital letters or capitalized words show the patterns	Each pattern is recognized by one class which is called "Sign"
$\supset$	Layer Inclusion	$l \supset m$ means layer l contains layer m	Package l contains package m
$\ni$	Pattern Membership	$l \ni P$ means layer l contains pattern P	Package l contains class P
,	Group Inclusion	$l \supset m$, n means $l \supset m$ and $l \supset n$	Package l contains both packages m and n
	Group Membership	$l \ni P$, Q means $l \ni P$ and $l \ni Q$	Package l contains both classes P and Q
.	Layer dependency	$l \supset m$. n means $l \supset m$, n and layer m is dependent on layer n, but layer n is not dependent on layer m	Package l contains both packages m and n, and package m has a dependency to package n, but package n has not a dependency to package m
$*$	Optional Layer	l^* means layer l is optional	Package l is optional
$?(c)$	Conditional Layer	$l^{?(c)}$ means existence of layer l is subject to condition c	Existence of package l is subject to condition described in c
$\{c\}$	*comment*	A *comment* can be attached to the above notations	*comment* explains the technical considerations

To enforce that "auxiliary layer" may include a sub-package "base," and if the designer decides, there could be sub-packages "distributed," "concurrency," and "sessionstate," we write the following rule. C1 is a predicate such as "designer wants Distributed Layer." C2 and C3 represent the corresponding predicates.

$$\textit{auxiliary layer} \supset \textit{base}^{*}\ ,\ \textit{distributed}^{?(\mathrm{C1})}\ ,\ \textit{concurrency}^{?(\mathrm{C2})}\ ,\ \textit{sessionstate}^{?(\mathrm{C3})}$$

To indicate which pattern resides in which layer, we use the pattern membership notation, for example, the following rule shows the placement of the patterns inside the Domain Layer. *domain* $\ni$ *Domain Model*, *Table Module*, *Transaction Script.*

II- Syntactic rules regarding the relationship between patterns

In Section 2.4.1, we discussed that patterns are not isolated. Patterns can be related to each other in different ways: uses, conflicts, refines, to name a few. Also we reviewed the literature and showed that there is no consensus among the pattern authors on the name, the meaning, the level of formality, and the formalism used for representing the relationships between the patterns of a PL.

Current PLs are not developed yet in terms of having clear and precise pattern relationships. Therefore, finding the the relationship between patterns in the *pattern form* is more difficult than finding the structural rules.

Following the compiler metaphor, the syntactic rules of a PL that show the relationship between patterns are similar to the grammar rules of a programming language that show how the tokens can be arranged to make a syntactically correct sentence. That means, the grammar of a PL must dictate the correct combination of patterns, considering the pattern relationships, that can be built based upon the patterns of the language.

Where to find syntactic rules regarding the relationship between patterns?

Finding the relationship between patterns in a PL is not a straightforward task. Sometimes this information is hidden between the lines of the prose text which describes the pattern. Sometimes, there is a very general and vague graph for representing pattern relationships. However, this information is not formal enough to be used as the basis for the operation of a PTV. In this section, we address possible cases that need to be considered in discovering the pattern relationships. In the next section, we propose a formalism for defining the pattern relationships.

The GOF authors mention some relationships between patterns:

> "Some patterns are often used together. For example Composite is often used with Iterator or Visitor. Some patterns are alternatives: Prototype is often an alternative to Abstract Factory" [GHJV95, p. 10].

However, the GOF book does not address the relationship between patterns in more detail. The "Related Patterns" field of the *pattern form* briefly talks about how patterns reference each other. Also, there is a general graph (see Figure 4 on page 32) which depicts the relationships for all 23 patterns in the language. Having such diagrams that show the "big-picture" of the language, and the relationship between patterns, is helpful but not sufficient.

In *PofEAA*, the relationships between patterns are not explicitly discussed by the author. The information is scattered in the texts that describe the patterns. More specifically the fields "Applicability" and "When to use it" discuss the relationship issues. In the introduction of the book, the author says "many of the patterns are about choices and alternatives" [Fow02, p. 6]. That means the book does not offer a single solution for an enterprise system. For every problem, there are many options, and it is the designer's job to make the trade off. A "Cheat Sheet" printed inside the back cover of the *PofEAA* book can be considered as an informal version of the grammar of *PofEAA* [Fow02].

In POSA-4, 114 patterns are grouped into 13 problem areas. There is a graph [BHS07a, p. 40-41] that shows important relationships between those 13 problem areas. Each problem area is described in a template, which contains a diagram that shows how the patterns are integrated into the PL. One of the problem areas, "From Mud to Structure," includes the root patterns of the language. The design process starts by selecting a root pattern, completing it with other patterns, and continuing until the designer arrives at one of the "leaf" patterns, i.e., patterns that can not be refined anymore by other patterns in the language [BHS07a, p. 40-41]. Despite the graphs for problem areas, and a template that is designed to show the relationships between patterns, POSA-4 authors use complementary explanations given in the prose text of the patterns.

Formalism for syntactic rules regarding the relationship between patterns

In Section 2.4.1, we presented several works that have addressed the relationship between patterns. Among them, we focus on three works which we found more comprehensive than others: James Noble [Nob98a], Wu-dong et al. [WdKqY$^+$03], and Buschmann et al. (POSA-5) [BHS07b]. By consolidating the idea of these three works, we define a notation in Table 8 for representing the relationships between patterns of a PL.

Table 8: Notations for Representing the Relationship Between Patterns

Notation		Meaning	In terms of UML
$\overline{P}$	root pattern	$\overline{P}$ means pattern P is a root pattern of the language, i.e., no other pattern is using P.	P is a mandatory class in the model, and no other class has dependency* to class P
$\longrightarrow$	uses	$P \longrightarrow Q$ means pattern P uses pattern Q	There is dependency* from classes of P to classes of Q
$\overset{cond}{\longrightarrow}$	conditional uses	$P \overset{cond}{\longrightarrow} Q$ means pattern P uses pattern Q subject to *cond*	There is dependency* from classes of P to classes of Q, subject to *cond*
$\mid$	alternative uses	$P \longrightarrow Q \mid R$ means pattern P may use pattern Q or pattern R	There is dependency* from classes of P to either classes of Q or classes of R
$\longleftrightarrow$	conflicts	$P \longleftrightarrow Q$ means patterns P and Q can not coexist in the model	The model can not contain both classes P and Q
$\overset{l}{\longleftrightarrow}$	conflicts in layer	$P \overset{l}{\longleftrightarrow} Q$ means patterns P and Q can not coexist in the layer l	Package l can not contain both classes P and Q
$\upharpoonright$	refines	$P \upharpoonright Q$ means pattern P is a specialized version of pattern Q	Class Q is a generalization of class P
$\{c\}$	comment	A *comment* can be attached to the above notations	*comment* explains the technical considerations
$\star$ Attribute dependency, method dependency, containment, or association.			

Note that every pattern combination has to start with one of the root patterns. A root pattern is an obligatory pattern and no other pattern is dependent upon it. The *uses* relationship is a basic relationship which can be found in most PLs. Three variants of *uses* are defined (*uses*, *conditional uses*, and *alternative uses*) to make it more usable. The *conflicts* relationship describes the situation where there is more than one solution to a specific problem, and those solutions are mutual exclusive. Two patterns can be conflicting either in the whole model or in a specific layer of the model. The *refines* relationship shows the case when one pattern is a more specialized version of another pattern. Finally, any comment that makes the relationship more understandable, especially from the technical and modeling point of view, will be given as a *comment.*

Examples of syntactic rules regarding the relationship between patterns

For starting a design with the *PofEAA*, a designer has several options as the initial pattern. For example, one may start from the domain, view, or controller. We select the last option, hence, either Front Controller or Page Controller could be the initial pattern. Then we need patterns for the View part of the system. Again, there are two alternatives, Template View or Transform View. This discussion, leads us to the following starting rules.

$\overline{\textit{Page Controller}} \rightarrow \textit{Template View} \mid \textit{Transform View}$

$\overline{\textit{Front Controller}} \rightarrow \textit{Template View} \mid \textit{Transform View}$

As another example of a syntactic rule in *PofEAA*, consider the following excerpt from the "How It Works" section of the Table Data Gateway pattern that reveals the relationship between this pattern and the Record Set pattern: "The trickiest thing about a Table Data Gateway is how it returns information from a query [...] you can return the Record Set that comes from the SQL query" [Fow02, p. 144].

The above text tells more than simple "usage" of one pattern by another. In fact, the text indicates "How x uses y?" This is interpreted as straightforward conditions, and is augmented to the *uses* rule. This interpretation should be performed by an expert in the domain of underlying PL. In the above case, since Table Data Gateway returns the data via its "find()" methods, a comment will be attached to the *uses* rule as indicated in the following. An implicit requirement of this rule is that "There should exist a Record Set pattern in the Base Layer of the model."

TableDataGatway $\rightarrow$ *RecordSet* {C4} , where C4: "The return type of every find() operation in the Table Data Gateway pattern is Record Set."

Another example in *PofEAA* is that for managing the transactional conflicts of business, there exist two patterns: Optimistic Offline Lock and Pessimistic Offline Lock. These two are in conflict if they are used for the same unit of work. The argument that there is a choice between these two patterns is made in the book as "The essence of the choice between optimistic and pessimistic locks is the frequency and severity of conflicts" [Fow02, p. 68].

This text can be interpreted as the following syntactic rule. The fact that the conflict happens when these two rules are applied on the same unit of work is explained via the comment line. $OptimisticOfflineLock \overset{concurrency}{\leftrightarrow} PessimisticOfflineLock$ {C5}, where C5: "The two patterns are applied for the same unit of work."

As an example of *refines*, we can consider both patterns Front Controller and Page Controller as refinements of Controller. (But note that there is no Controller pattern in *PofEAA*, and Fowler [Fow02, p. 56] prefers to call the controller part of the Model View Controller, the input controller.) Therefore, we can define the following syntactic rules: $FrontController \upharpoonleft Controller$ and $PageController \upharpoonleft Controller$.

3.3.3 Semantic Rules

The PMV phase uses the semantic rules to verify whether or not a pattern combination used in the design is semantically correct. To the best of our knowledge, there has been no discussion on the semantics of a pattern combination in the PL community. Even in compiler design, the semantic checking is considered an optional phase, such that, some compilers perform no semantic analysis at all; Other compilers limit it to type checking or code generation issues [ASU86]. For instance, a syntactically correct "assignment statement" might be considered as having semantic error if there is type mismatch between the types of both sides of the assignment.

In this thesis, we consider two categories of semantic problems. The first category are the conflicts between the applied patterns and the context information. That means, we consider the context of design as a parameter which affects the semantics of the design. By context information, we mean any information which is related to the system environment. The following list shows some examples of the context information: implementation language, expertise level of the designer, underlying DBMS technology, and possibility of transaction conflict.

The semantic rules should clearly say which pattern is in conflict with which context information. For instance, if applying a pattern is not recommended for a novice designer, or applying a pattern does not match well with the implementation tool or the DBMS, these facts must be reflected in the semantic rules related to either of those patterns.

The second category of semantic problems are the inconsistencies between the features of applied patterns. The semantic rules must prevent any conflict between the features (behavioral or structural) of 1) a single class in a pattern, 2) the constituting classes of a single pattern, or 3) the classes of different patterns. For instance, requiring getters and setters for the attributes of a class, consistency between the attributes of two cooperating classes in a pattern, and consistency between the operations of two corresponding patterns are examples of semantic rules that must be adhered by the designers. Semantic rules must be precise enough to catch such errors.

Where to find semantic rules?

We were unable to find any PL which has explicitly addressed semantic issues of the language. Thus, discovering the semantic rules of a PL is even more difficult than the syntactic rules, since the information, if any, is again hidden in the prose description of the patterns. Note that syntactic rules and semantic rules are more PL-oriented than the structural rules. It would be beneficial if the pattern authors dedicate a particular field in the *pattern form* to address the semantic issues.

Formalism for representing semantic rules

We define two general notations in Table 9 for describing the inconsistencies between a pattern and the context information, or the inconsistencies between the features of a combination of patterns. Note that these two notations are complementary. The criteria that cause inconsistencies for the pattern must be written clearly and precisely in the condition part of the rule.

Table 9: Notations for Representing the Semantic Rules of a PL

Notation		**Meaning**	**In terms of UML**
$\approx$	consistent	$P \approx \{c\}$ means pattern P is consistent with the condition specified by $\{c\}$	Class P can not exist in the model while the condition in $\{c\}$ is violated
$\not\approx$	inconsistent	$P \not\approx \{c\}$ means pattern P is inconsistent with the condition specified by $\{c\}$	Class P can not exist in the model while the condition in $\{c\}$ is hold

Examples of semantic rules

In this section, we present some recommendations given in *PofEAA* book, and show how they can be interpreted as semantic rules.

Tool Consistency The following excerpt from the book explains the consistency between the Table Module and the development environment (tool).

> "If you have an environment like .NET or Visual Studio, then that makes a Table Module much more attractive" [Fow02, p. 30].

This can be interpreted as a semantic rule of the first category, i.e., consistency between a pattern and context information: $\mathit{TableModule} \approx \{\mathit{Tool} = .\mathit{NET}\}$

Operation Parameter Consistency In the Table Data Gateway pattern, it is claimed that:

> "the parameter list of the insert method must be a subset of the parameter list of the update method" [Fow02, p. 144].

The semantic rule corresponding to this claim belongs to the second category. This rule can be written as:

$\mathit{TableDataGateway} \approx \{\mathit{insert}()\ \mathit{parameter\ list} \subseteq \mathit{update}()\ \mathit{parameter\ list}\}$

3.4 PLV vs. Compiler

We explained that there is an analogy between the tasks of the PLV process and what a compiler does. As a compiler has phases for checking the lexical, syntactic, and semantic aspects of a programming language, PLV also has phases for verifying the structural, syntactic, and semantic aspects of a PL. Both processes use three groups of rules as touchstones for judging about the lexical, syntactic, and semantic aspects of the language. This similarity has also been identified by other researchers and pattern pioneers. For instance, Buschmann et al. in [BHS07b] discuss how a PL needs a grammar for guiding the designer in building acceptable pattern combinations.

Despite the similarities, there exist several differences between the PLV process and the compiling process.

1. It is widely accepted in the programming languages community that the lexical, syntactic, and semantic rules of a new language must be defined precisely and formally, to enable us to build a compiler for that language. However, in PLs, formality, if it exists, is used mainly for describing the structure of a pattern, and the rules that define the pattern relationships and best practices of the language, are mostly written in natural language. That means, building a PLV for a PL needs extra steps of formalizing the rules that govern the PL in order to make them ready for the PLV modules.

2. In compilers, a source program usually consists of tokens from the source language: any other thing is reported as an error. In a design model, which is given as input to

the PLV process, in addition to the patterns that belong to the underlying PL, there could be other patterns or model elements which do not belong to the underlying PL. The PLV process just ignores those elements.

3. In the PLV process, all the modules work on the same model and we cannot say that each phase changes the model from one representation to another, as it happens in the phases of a compiler.

4. While a compiler consists of both analysis and synthesis, PLV deals only with the analysis. That means, we verify the design model and try to fix the design problems, but, we do not attempt to generate code from the model.

5. Most compilers do not allow a program to compile until all the errors are fixed. However, PLV is not an intrusive process, i.e., detecting an error does not impede the designing process. The designer always has the choice to ignore an error completely, or to fix it later.

6. A compiler converts a program from a source (high-level) language to a target (low-level) language. In PLV, the source and the target are both models in the same level, only some of the errors in the source model may have been fixed.

3.5 The Profile-driven PLV Process

The PLV process presented in Section 3.2 is a simple three-phase process that verifies a UML design model from the structural, syntactic, and semantic viewpoints of the underlying PL. In this section, we explore the role of a UML profile for a PL in the PLV process. The aim is to present a more elaborate PLV architecture and clarify the responsibilities of each module. In Section 3.5.1, we give an overview of the changes to the simple PLV. The new architecture of the PLV will be presented in Section 3.5.2. The four main modules of the new architecture will be explained in Section 3.5.3 to Section 3.5.6 respectively.

3.5.1 Overview

Three new features are added to the PLV process include adding a module for helping the designer in fixing the problems, bookkeeping the pattern information by the earlier modules to facilitate the task of next steps, and utilizing a profile to ease the pattern detection and accessing configuration information. These features are explained in the following sections.

Fostering PLV with Advisory Power

In order to add the advisory power to the PLV, we add a new module called **Pattern Language Advisor (PLA)** to its structure. PLA is responsible for reporting the errors to the designer, displaying guidelines on how to fix the problems, fixing the detected problems in a systematic manner, and recording the model modifications into a **Design Rationale**.

Upon detection of an error in the model, PLA is invoked, and having access to all the structural, syntactic, and semantic rules, guides the designer in stepwise fixing of the problems. PLA also gives the designer the opportunity for systematic repair. Therefore, PLA is the only module which is able to apply modifications to the model. For more details see Section 3.5.6.

Pattern Information Table (PIT)

Influenced by the idea of symbol table in compiler design, we have a table which tracks information of detected patterns, called the **Pattern Information Table (PIT)**. The PIT is created by the PSV and contains information about the detected patterns, e.g., pattern name, the layer in which the pattern is placed, and the pattern elements. Pattern elements form a list that shows the actual parameters assigned to the formal parameters of a pattern. The PTV, PMV, and PLA modules will use this table to know which patterns are detected in the model and in which layer they are placed.

Using a UML Profile in PLV

We concur with Martin Fowler that "The biggest software patterns community is rooted in the object-oriented world" [Fow03]. Furthermore, the initiative of PLs in software has started from the OO discipline (OOPSLA [OOP09] and PLoP [Hil09a] conferences), and UML is the dominant modeling language for OO systems [Lan06]. After a designer built a model based on the patterns of a PL, it is not always clear what patterns are used in the model, without having some metamodel-level information about the model. For instance, none of the related work introduced in Chapter 2 are able to detect all GOF patterns. A profile is an extension mechanism for UML, which allows us to customize UML, for example, by extensions representing the PL elements.

The above facts encourage us to utilize the UML profile mechanism in making the PLV process more effective. This will change the simple architecture of the PLV shown in Figure 13 to a profile-driven process. That means, all the modules utilize a UML profile that should already be defined for the underlying PL. We call such profile a **Pattern Language UML Profile (PLP)**.

A profile is defined by specifying three sets: Stereotypes, Tagged Values, and Constraints. Stereotypes are concepts from the domain that are defined to extend one of the existing UML meta-classes. For each stereotype, tags can be defined to save configuration information. These tags act as meta-attributes for the corresponding stereotype. Values can be assigned to tags to make "tagged value" pairs. Constraints are the Well-Formedness Rules (WFRs) defined for the stereotypes. Applying a stereotype on a model element, causes the WFRs of that stereotype to be verified.

In the following sections, we will elaborate on the information that is captured by each element of the profile, and how these elements help the PLV modules.

Stereotypes

For the PLV process, stereotypes are the most important elements of the profile, because they provide a naming convention for each pattern. The designer uses the stereotypes for the following purposes.

- To indicate the pattern he/she wants to apply: We suppose that for each pattern there is a unique class stereotype. This stereotype acts as the "Sign" for the pattern and releases us from the pattern detection endeavor.
- To name the constituent elements of a pattern: Every element of a pattern (class, attribute, or method) has an appropriate stereotype.
- To indicate the layer containing the pattern: Stereotypes are defined for packages that show the layers in a layered architecture.

All PLV modules utilize stereotypes. In searching for the constituent elements of a pattern, PSV can directly find the element. The PTV module utilizes stereotypes when looking for the containing layer of a pattern or in checking the dependencies between patterns. The PMV module uses the stereotypes in finding a specific feature (attribute or operation) of a pattern. PLA should attach the corresponding stereotypes to the elements that are added to the model.

Tagged values

Tagged values are helpful for capturing configuration/context information of the model, e.g., the implementation language. Tagged values allow the designer to define values for tags dynamically during the design and the values are persisted with the model. If we do not utilize tagged values, capturing context information must be done using auxiliary files or via the Graphical User Interface (GUI) of the modeling tool, however, neither of these approaches gives information which is synchronized with the model.

Both PSV and PTV use tagged values to access context information that is required for verifying some of the structural or syntactic rules. PMV is the main user of the tagged values, since this module is responsible for checking the inconsistencies between the used patterns and the context information.

Constraints

Constraints are WFRs that are defined for each stereotype of a profile. Applying a stereotype on a model element, causes the WFRs of that stereotype to be verified. For building a PLP, we must have three groups of constraints: structural, syntactic, and semantic. These constraints are the basis for the operation of PSV, PTV, and PMV respectively.

There are two main alternatives for defining the constraints of a profile:

1. Informally by a natural language, in which case, the constraints must be hard coded by a programmer in order to build the three verifier modules of PLV.

2. Formally by Object Constraint Language (OCL), in which case, the three verifier modules of PLV are in effect applying the profile on a model and verifying the OCL constraints.

As we discussed in Section 3.3, the formalisms we proposed for PLV rules contain many textual comments which makes them far from being easily translated into a formal language such as OCL. Therefore, our strategy is to select the first alternative and hard code the rules into the three verifier modules.

It should be noted that the profile constraints are not meant to perform model modifications which are the duties of PLA. Model modifications can be simple, such as adding a missing operation to a class, or complicated, such as building an instance of a pattern automatically, which is called "pattern instantiation."

To summarize, PLV as a profile-driven process: The profile plays an important role in the PLV modules, since these modules make use of the stereotypes and the tagged values, however, the constraints of the profile need to be hard coded by the programmer who is building the PLV.

3.5.2 PLV Architecture

The extended architecture for the PLV process is shown in Figure 17 [ZBK09].

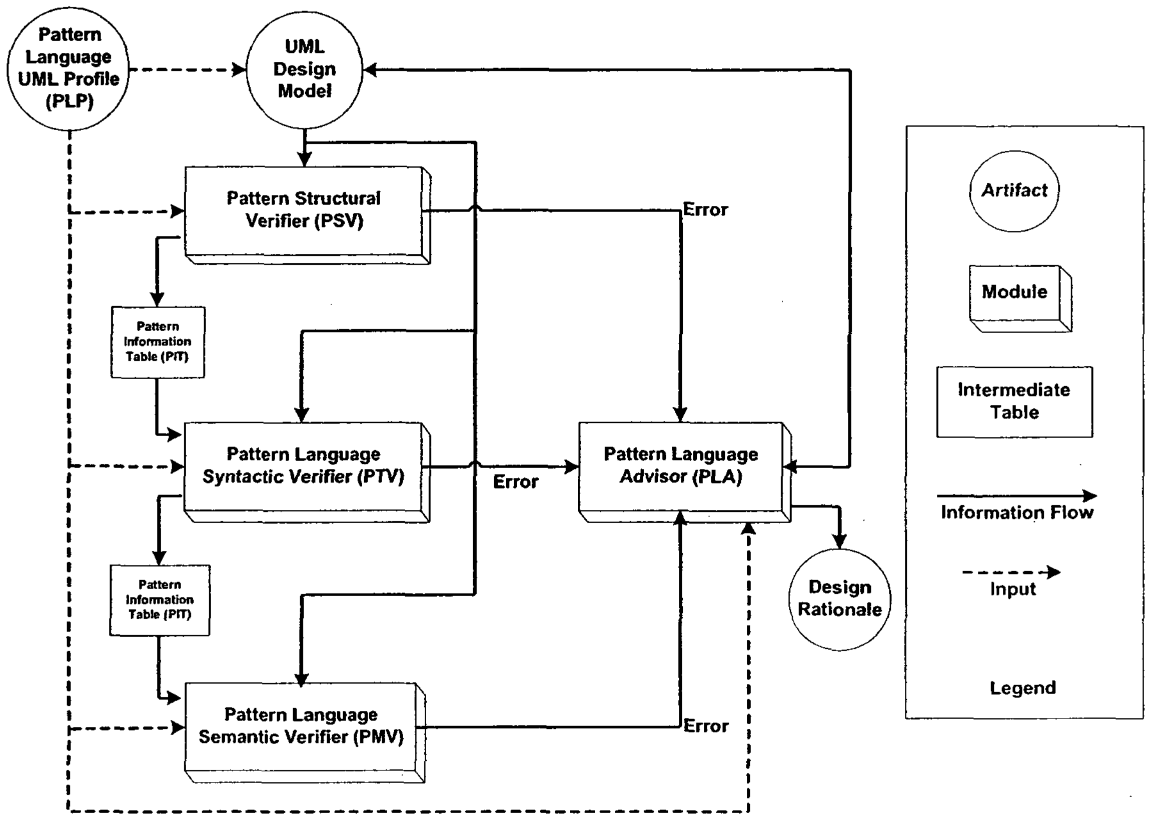

Pattern Language UML Profile (PLP)
UML Design Model
Pattern Structural Verifier (PSV)
Error
Pattern Information Table (PIT)
Pattern Language Syntactic Verifier (PTV)
Error
Pattern Language Advisor (PLA)
Pattern Information Table (PIT)
Design Rationale
Pattern Language Semantic Verifier (PMV)
Error
Artifact
Module
Intermediate Table
Information Flow
Input
Legend

Figure 17: The PLV Architecture

The process deals with two artifacts: UML Design Model and Design Rationale. The UML Design Model is the input information to all four modules; This is shown by solid directed lines that go from the model to the modules. The Design Model is also an output of the process, due to the modifications that the PLA may apply on it, hence the solid line from the model to the PLA is directed at both ends. The other artifact is the Design Rationale which is an output text file recording the changes made to the model by the PLA, thus, a solid directed line goes from PLA to the Design Rationale.

The architecture also reveals that the process is profile-driven, since both the Design Model and the modules utilize the profile by using the stereotypes and tagged values. This fact is shown by the directed dashed lines from the PLP to both Design Model and the modules. The PIT records information about the detected patterns, obtained by PSV, and forwards this information to the next phases. In the following sections, we describe the responsibilities of four main modules of the process.

3.5.3 Pattern Structural Verifier (PSV)

PSV accepts the Design Model as input and, by verifying the structural rules of the PL, looks for single patterns that are correctly applied in the model. The designer shows his/her intention of applying a particular pattern by using the "Sign" stereotype of that pattern on one of the classes in the model. By detecting the "Sign," PSV initiates the verification process to check the structure of the pattern. If the correct usage of the pattern is detected, the information about that pattern, is recorded into the PIT. If errors are found in the structure of the pattern, the PLA is invoked to report the errors and help the designer fix the problems.

PSV applies the "structural match" strategy. That means, it matches the structure of the pattern given in the Design Model, with the structure of the pattern that is defined by the structural rules. For doing this task, PSV applies ideas introduced by the Sign/Criteria/Repair (SCR) process, except the repair part [ZB07, ZKB08]. The matching process starts from the "Sign" of the pattern. When the "Sign" is found, PSV initiates verifying the "Criteria" of the pattern. "Criteria" contains a set of structural rules (constraints) which defines the correct application of the pattern. PSV navigates the associated model elements and checks for the validity of the constraints in the "Criteria." Then, it traverses the associated pattern elements (classes) based on the structural rules. For each class, the features (attributes and operations) are also checked against the structural rules. If all the rules are satisfied, the pattern is detected correctly, and is recorded into the PIT.

Leveraging a unique "Sign" for each pattern, we eliminate the possibility of ambiguity in detecting patterns that have similar structure, or patterns that are part of another pattern. For instance, consider patterns State [GHJV95, p. 305] and Strategy [GHJV95, p. 315] from GOF design patterns that are not easily distinguishable since their structure is very similar [NB02]. Without a naming convention, or a utility such as stereotype, it is impossible to detect each of these patterns. Note that none of the GOF design pattern detection methods discussed in Chapter 2 are able to detect both of these patterns unambiguously without considering some context information or dynamic views of the model.

PSV may encounter a variety of structural errors during the verification of patterns. Following is a list of possible errors with some appropriate examples on the Front Controller pattern (See Figure 8 on page 40). Note that "pattern element" means any part of the structure of a pattern including a class, an operation, an attribute, or an association.

- Missing element in a pattern, e.g., missing the "process" operation in the Command class.

- Missing part of a pattern, e.g., missing the "Command" part.
- Incorrect or missing relationship between two pattern elements, e.g., missing dependency between Handler and Command classes.
- Incorrect property of pattern elements, e.g., Command class which is not abstract.
- Incorrect cardinality of pattern elements, e.g., having less than one Concrete Command class.

3.5.4 Pattern Language Syntactic Verifier (PTV)

PTV verifies the model based on the syntactic rules of the PL. This verification includes both checking the layering of patterns and checking the relationships between the detected patterns. Therefore, there are two types of errors that can be caught by PTV. First, placement of a pattern in a wrong layer or group. Second, a missing relationship between two patterns.

During the course of action, PTV uses PIT to find the detected patterns, their layers, and their constituent elements. PTV updates the layering information of each pattern into the PIT, and inserts new information about the pattern relationships in this table. In case of error, PTV invokes PLA to report the error and guide the designer in fixing the problem, either manually or systematically by PLA.

Some of the syntactic errors are simple and can be easily caught by merely querying the PIT. Some examples are:

- Inconsistency between a pattern and its containing layer or group. When a pattern is placed in an inappropriate layer, PTV detects it easily by checking the table. For instance, in *PofEAA*, placing a pattern which belongs to the Data Source layer (e.g., Table Data Gateway) in the Domain Layer will result a syntax error.
- Conflict between the patterns in a layer or group. When two inconsistent patterns are placed in a layer, the error can be caught by checking the table. For instance, in *PofEAA*, applying both Optimistic Offline Lock and Pessimistic Offline Lock patterns for resolving concurrency issues for the same unit of work in the design will trigger a syntax error.

However, more complicated syntactic errors may need model investigation in addition to accessing the PIT. Some examples are:

- Incorrect relationship between two patterns, e.g., in *PofEAA*, having Transaction Script in the Domain Layer, and then having both Table Data Gateway and Row Data Gateway in the Data Source Layer contradicts the alternative relationship between those two and cause an error.

- Missing relationship between two patterns, e.g., in *PofEAA*, having both Table Module in the Domain Layer and Table Data Gateway in the Data Source Layer where both are accessing the same data, and there is no uses relationship between them will result in a syntax error.

In case of any error or inconsistency, a call to the PLA is made in order to report the problem to the designer and assist him/her repair the problem. Some of the syntactic errors can be fixed automatically by the PLA, but most cases need designer's decision and manual modifications on the model. As an example of the former case, consider the situation that the designer has used the Domain Model pattern and, for one of the domain objects, a Data Mapper is needed in the data source layer. Creating the Data Mapper can be performed automatically upon the designer's request by the PLA. As an example of the latter case, detecting an error due to having both Table Data Gateway and Row Data Gateway for accessing the same data can not be resolved automatically, since it needs designer's decision. In both cases all the changes to the model are applied after the designer's confirmation.

3.5.5 Pattern Language Semantic Verifier (PMV)

The PMV module is responsible for verifying that the model adheres to the best practices of the PL. Specifically, by applying the semantic rules of the PL, PMV verifies that the model is consistent with the context information of the system. Examples of the context information are: the implementation language, the designer's expertise, the designer's choice for optional patterns, and the complexity of the system.

If any inconsistency is found in the applied pattern combination, the PLA is invoked to report the errors and help the designer fix the problems. Many of the problems are easily fixed by setting the appropriate value for the context information, e.g., selecting another tool for implementation. These repairs can be done automatically by the PLA, subject to the designer's confirmation. Other problems that are solved only be changing the applied pattern, should be solved manually by the designer.

The following list shows some of the possible errors that PMV can recognize, with examples from *PofEAA*.

- Discrepancy between the context environment and the choice of patterns. For instance, *PofEAA* suggests that if a machine is controlling the screen flow of the system, then we need an Application Controller pattern, otherwise, we do not need it [Fow02, p. 58]. Now, if we have such information that the screen flow is not machine-controlled (such information can be obtained from tagged values), and the Application Controller pattern is detected in the Presentation Layer of the model, a semantic error is triggered.

- Inconsistency between pattern elements. For example, there is a semantic rule for the Table Data Gateway pattern which enforces that the parameter list of the "insert" operation be a subset of the parameter list of the "update" operation.

3.5.6 Pattern Language Advisor (PLA)

PLA is an important module of the PLV process, which is responsible for reporting the errors, displaying the guidelines, and helping the designer fix the problems. Reporting the errors and displaying the guidelines are important steps that foster a novice designer's knowledge in learning more about the patterns and PL. Fixing the problem might be done automatically by the PLA, or manually by the designer.

For the cases that PLA is able to perform automatic repair, it gives the suggestions to the designer, and by the designer's request, the required modifications are applied to the model. Following are some of the modifications that are doable automatically by the PLA.

Pattern Instantiation Although this is not meant to be the main responsibility of PLA, but it can be achieved indirectly. For instantiating a pattern, the designer applies only the Sign stereotype of the pattern on a class and leaves the completion of other parts to the PLA. This way, PSV finds the structural errors and invokes the PLA to fix the problems. Then, PLA will add missing elements and relationships to the model such that the pattern is applied correctly. Another way of instantiating a pattern, is when one pattern needs another pattern in a *uses* relationship, and the PTV catches the error. Then, the latter pattern can be instantiated automatically by the PLA. A straightforward example is when the designer has applied the Table Data Gateway pattern, and the return type of the find operation in that pattern needs to be the Record Set pattern. Hence, the Record Set can be automatically set as the return type of the find operation, and then the pattern must be created in the Base Layer of the model.

Adding missing elements to a pattern A missing element could be an element of a single pattern, e.g., a class, an attribute, a method, or a relationship between the constituent classes of the pattern; These kind of errors are reported by the PSV. Or, a missing element may belong to the model, e.g., a package, a relationship between patterns. Such errors are reported by the PTV. In both cases, the PLA can fix the problem automatically, by adding the missing elements to the model.

Changing the properties of an element For instance, PLA changes a non-abstract class to abstract in order to correct the application of a pattern, and fix the error which was caught by the PSV.

Changing the dependency between the elements of a pattern Examples are changing the cardinality, navigability, or containment of an association between constituting classes of a pattern. Such errors may have been caught by the PSV.

For problems that are hard to fix automatically or need expertise or designer's decision, the guidelines for fixing the problem should be given to the designer, and it is the designer's responsibility to modify the model accordingly. However, providing the designer with guidance and supporting comments can expedite the error recovery process. For such cases, PLA helps the designer by displaying useful guidelines based on the PL which shows the roots of the error, the rationale behind the error, and the reference to the technical details on how to fix the problem.

As an important job, PLA records all the modifications that are automatically made to the model, in a Design Rationale document. The Design Rationale is a document that shows what issues have been investigated about the model, what alternative solutions have been considered, which one is selected, the justifications behind the decisions, along with the modifications that has been made into the model. Design rationale is a fruitful document for the system maintainers [PB88].

3.6 Discussion

3.6.1 Summary

Inspired by the compilers, a process for verifying the use of a Pattern Language (PL) in a design model is presented. The process is named Pattern Language Verifier (PLV) and consists of four modules (phases): three verifier modules (PSV, PTV, and PMV) that verify a design model from the structural, syntactic, and semantic points of view, and an advisor module (PLA) that helps the designer repair the problems. If possible, the advisor module repairs the problem automatically. All the automatic modifications are recorded into a Design Rationale to be used by the designer in understanding the evolution of the design. A model is structurally correct if all the patterns are applied correctly. Syntactic problems are related to the layering and relationship of patterns. Semantic issues are mainly the inconsistencies between the choice of patterns and the context information.

As the touchstone, the verifier modules use the structural, syntactic, and semantic rules of the underlying PL. It is shown how and where to extract the rules of a PL, then new formalisms for representing the rules are introduced. The formalism for structural rules is very simple, just a mixture of class diagrams with clear English sentences. However, the formalism for syntactic and semantic rules is more precise, and is inspired by the Context-Free Grammar (CFG) notation.

PLV is a profile-driven process. A prerequisite for the process is to define a UML Profile for the PL. The profile includes stereotypes, tagged values, and constraints. The stereotypes play the identifying role for the patterns and their elements. Tagged values are used to save information about the context of the design. Constraints are indeed the three groups of *structural, syntactic*, and *semantic* rules. The constraints can be translated into Object Constraint Language (OCL), or kept in the formalism that we proposed. We recommend the latter case, since OCL is not meant to perform modifications on a model. Hence, the PLV modules must be hard coded into a tool to be used by a designer.

3.6.2 Possible Extensions to the PLV Modules

Extensions to PSV Our assumption is that *all* the structural criteria of a pattern shall be satisfied in order to cause PSV report the correct application of that pattern. However, there are cases that some of the criteria are not as fundamental as others and they can be ignored. As an extension to this module, one can add a "severity level" parameter that acts as a threshold for the sensitivity of detecting a correct pattern. This way the PSV is able to report "near-misses" for each pattern. A near-miss of a pattern means a structure which is very close to the structure of a pattern, but it has minor deviations. Reporting near-misses is quite educational for novice designers.

Another possible and desired extension to the PSV is to investigate also the dynamic views of the model, e.g., the UML sequence diagrams. As discussed in Section 2.5.2, for some of the patterns, merely static checking is not enough, e.g., for the Singleton pattern.

Extensions to PTV For checking pattern layering, currently we only check that the "Sign" of a pattern is in the appropriate layer. It is wise to check that all the elements of the pattern reside in that layer.

Another extension to the PTV is to check the inter-collection rules. Currently, we only investigate the intra-collection relationships between patterns. It is true that most of the problems can be solved by adding more rules to the grammar, however, the cases such as conflicting pattern names must be handled carefully.

Extensions to PMV Adding linguistic knowledge to PMV enables it to detect errors such as the violation of advice A21 in Appendix A.2 which recommends that if we use Domain Model pattern, the name of the domain concepts should be selected among the nouns in the domain [Fow02, p. 26].

3.6.3 Pattern Language Issues

Current PLs are not mature yet in terms of having clear and precise pattern relationships. Therefore, finding the relationship between patterns in the *pattern form* is more difficult than finding the structural rules. There are several reasons for this problem as follows.

1. Despite the structural rules that deal with one pattern and, most of the time, can be found within specific fields of the *pattern form*, the syntactic rules for pattern relationships are scattered across the engaged patterns and need to be extracted by investigating those patterns.

2. Most of the PL authors prefer to present the syntactic rules in prose text, interwoven with the pattern descriptions. The difficulty of extracting syntactic rules depends on the level of formalism that is used for showing the pattern relationships.

3. The work on formalizing the grammar of a PL is sparse. Our literature survey (see Section 2.4.1) showed that such endeavors are in their infancy stages yet. For instance, in POSA-5 [BHS07b], as one of the recent works in this area, it is tried to give a grammar-like formalism for the syntax of a pattern sequence. But, we think POSA-5 is still immature, as the authors also emphasize that "not all the aspects of pattern languages we discuss in this part of the book are mature or well-established in the pattern community. [...] aspects and properties, such as the role of pattern sequences in defining a grammar for pattern languages, are considered as new or even subject to debate" [BHS07b, p. 245].

3.6.4 Profile Issues

PLV is a profile-driven process, hence, defining a UML profile for the underlying PL is inevitable. An important issue is to investigate "to what extent does a UML profile suffice for fulfilling the tasks of the PLV process?"

Utilizing profile (particularly its stereotypes) reveals us from the vexing problem of pattern detection. That is because designers explicitly announce which patterns are used in the model by using the "Sign" of the patterns. Without profile, we need to apply one of the pattern detection strategies reviewed in Chapter 2, which for a sample pattern collection such as GOF, none of them are 100% capable of detecting all the patterns in that collection. Remember that there are outwardly similar patterns that distinguishing them only from their structure is almost impossible.

Furthermore, utilizing tagged values is an easy way to access meta-data such as configuration/context information; This information is up-to-date and is orchestrated with the model since it is acquired during the design. Without tagged values, accessing such data my need reading offline files or extending tool's GUI, which is more tedious and is not synchronized with the design.

We believe that even if the constraints are written in OCL and the three verifying modules are inherently built, the PLA must be built explicitly, since the duties of PLA are out of the scope of profile abilities. That is because the profile constraints are not intended to perform model modifications. Another important issue with using OCL, is the lack of persistent data between the constraints. Hence, when a structural constraint verifies the structure of a pattern in the model and ensures that the pattern is applied correctly, the detected pattern's information (i.e., the PIT) must be persisted somewhere that can be accessed by the PTV. One solution to this issue, is to check all the structural, syntactic and semantic criteria of each pattern all together in the constraints of that pattern. In this approach, the syntactic rules do not limit themselves to the correctly applied patterns. However, one problem to this approach is that for rules like "Pattern A *uses* Pattern B," it is unclear in which pattern this rule must be verified, Pattern A or Pattern B?

Chapter 4

A Pattern Language Verifier (PLV) for *PofEAA*

This chapter shows which steps should be taken in order to make a Pattern Language Verifier (PLV) for a Pattern Language (PL). For our case study, we have selected Patterns of Enterprise Application Architecture (*PofEAA*) [Fow02] PL. We became familiar with *PofEAA* in Section 2.6. *PofEAA* consists of 51 patterns, however, we have selected a subset containing the 23 patterns we need for our case study. As an environment in which we have hard coded the PLV modules, we have selected the ArgoUML modeling tool. The resulting tool, which is a "PLV for *PofEAA*," is called ArgoPLV.

The remainder of this chapter is organized as follows. In Section 4.1, we introduce the selected patterns and their relationships. Section 4.2 discusses the advices that form the structuring mechanism of the *PofEAA* PL, and shows how these advices are formalized as the formal rules for the PLV modules. In Section 4.3, we introduce the "*PofEAA* UML Profile" as an important component required by the PLV process. Section 4.4 shows how the "PLV for *PofEAA*" is built as a plugin for the ArgoUML, which is called ArgoPLV. Section 4.5 discusses what has been learned from our case study.

4.1 *PofEAA* Selected Patterns

As we have introduced in Section 2.6, *PofEAA* consists of 51 patterns categorized into three main layers and seven supporting layers. For the sake of simplicity and concreteness, we selected a subset of *PofEAA* that contains 23 patterns from several layers. These are the patterns we need for our case study.

Patterns that are filtered out are mainly "Object-Relational" patterns that deal with mapping classes to the tables of a relational database. Although they are important patterns for a practical enterprise application, these patterns are more database-related. From the PLV perspective, considering them does not add any knowledge to codifying process that we are going to present in this chapter. There are sixteen patterns in the "Object-Relational" category that are excluded in this case study. In addition, we have excluded one of the Session State patterns, which stores session data in the database. Furthermore, we have excluded six patterns from the Base Layer, two patterns from the Offline Concurrency Layer, and three patterns from the Presentation Layer. In total, 28 out of 51 patterns are excluded in this case study.

Table 10 shows the number of patterns that are selected and the number of patterns that are excluded from *PofEAA* in the case study. Table 11 shows the name of the patterns that are selected or excluded from each layer. Note that in *PofEAA*, "Service Layer" is the name of a pattern in the "Domain" Layer, however, we dedicate a separate layer for this pattern. That means, we have a "Service Layer" pattern in the "Service" Layer.

Table 10: Statistics on Selected and Excluded Patterns from *PofEAA* in our Case Study

Layer/Category	**Patterns**	**Selected**	**Excluded**
Presentation	7	4	3
Service	1	1	0
Domain	3	3	0
Data Source	4	4	0
Object-Relational Behavioral	3	0	3
Object-Relational Structural	10	0	10
Object-Relational Metadata Mapping	3	0	3
Distribution	2	2	0
Offline Concurrency	4	2	2
Session State	3	2	1
Base	11	5	6
Sum	**51**	**23**	**28**

Figure 18 shows the placement of selected patterns in a layered architecture. In addition to the eleven patterns in the three main layers (presentation, domain, and data source), we have one service pattern, five base patterns, two concurrency patterns, two session state patterns, and two distributed patterns in their respected layer. Hence, 23 patterns are shown in the figure. General descriptions of the layers and the patterns of *PofEAA*, given in Section 2.6, are still valid and useful. In this section, we elaborate on the patterns in Figure 18 and their dependencies.

For designing the architecture of a web-based enterprise application, the designer may start from the Presentation Layer of the system. Taking the Model View Controller

Table 11: Selected and Excluded Patterns from *PofEAA* in our Case Study

Layer/ Category	**Patterns** (Bold means Selected)
Presentation	**Page Controller, Front Controller, Template View, Transform View,** Model View Controller, Two-Step View, Application Controller
Service	**Service Layer**
Domain	**Transaction Script, Domain Model, Table Module**
Data Source	**Table Data Gateway, Row Data Gateway, Active Record, Data Mapper**
Base	**Layer Supertype, Money, Record Set, Gateway, Mapper**, Separated Interface, Registry, Value Object, Special Case, Plugin, Service Stub
Object-Relational Behavioral	Unit of Work, Identity Map, Lazy Load
Object-Relational Structural	Identity Field, Foreign Key Mapping, Association Table Mapping, Dependent Mapping, Embedded Value, Serialized LOB, Single Table Inheritance, Class Table Inheritance, Concrete Table Inheritance, Inheritance Mappers
Object-Relational Metadata Mapping	Metadata Mapping, Query Object, Repository
Distribution	**Remote Facade, Data Transfer Object**
Offline Concurrency	**Optimistic Offline Lock, Pessimistic Offline Lock**, Coarse Grained Lock, Implicit Lock
Session State	**Client Session State, Server Session State**, Database Session State

paradigm, the designer needs patterns for the Controller part and the View part. There are two alternative patterns for the Controller part, the Front Controller and the Page Controller, which their selection depends upon the implementation environment and the simplicity of the requests. The choice for the View part is the corollary of the Controller and the tool selection. That means, either the Transform View or the Template View can be used with either of the Controllers depending upon the tool.

The next step, in designing a web-based enterprise application, is to select patterns for the Domain Layer. There is no specific dependency between any of the patterns in the Presentation Layer and the layer beneath. However, if the designer decides to use the Service Layer pattern as an API for the application, this pattern usually works with a Domain Model or Table Module.

An important decision is the selection of a pattern for the Domain Layer of the system. If the designer is looking for an easy and straightforward solution, the Transaction Script pattern is the choice. However, for complicated systems which have a lot of domain concepts, and when Object-Oriented is able to better describe the structure of the domain, the Domain Model pattern is an appropriate selection. For intermediate situations, i.e., when the business logic is not too complex, the use of the Table Module pattern is recommended.

Selecting patterns for the Data Source Layer is more dependent upon the Domain Layer patterns. If a Domain Model pattern is selected and the domain is rich, i.e., the structure of the domain model is complex, then the suggested pattern for the Data Source Layer is Data Mapper; Otherwise, Active Record is a better choice. Note that according to the *PofEAA*

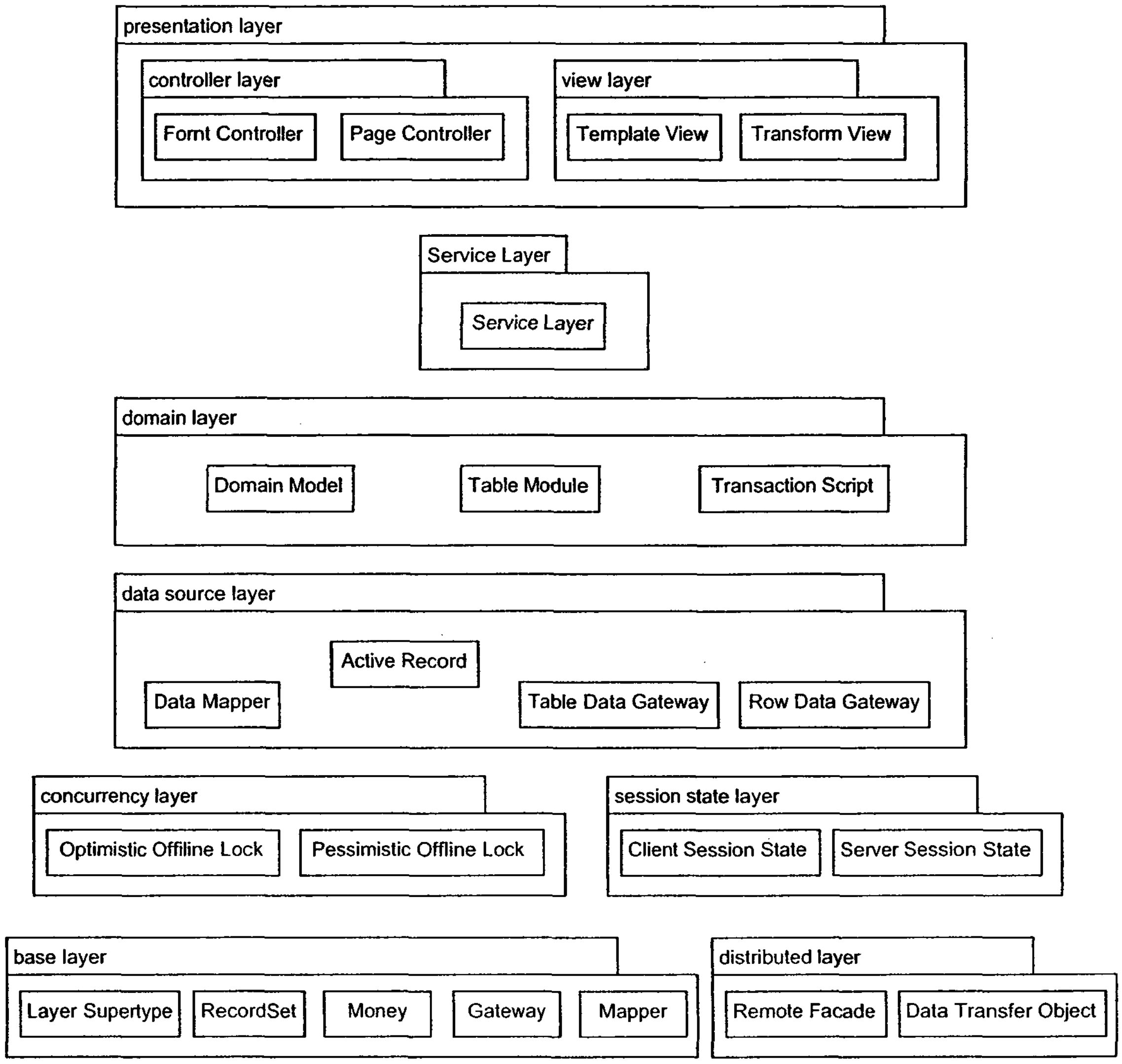


Figure 18: Selected Patterns from *PofEAA* in a Layered Architecture

book, Active Record is in the Data Source Layer, therefore, in Figure 18, we kept Active Record inside that layer. However, to reflect the fact that it may contain some business logic, it should be considered on the boundary of the Domain Layer and the Data Source Layer. In case a Transaction Script or Table Module is selected, then there are two options for the Data Source Layer: Table Data Gateway or Row Data Gateway.

Finally, there are optional layers regarding the concurrency, storing session data, or distributed issues. For handling the conflicts that occur in concurrent sessions, there are two options in the Concurrency Layer. If the chance of conflict is high, the Pessimistic Offline Lock pattern is the right choice, otherwise, the Optimistic Offline Lock pattern suffices. For storing session data, there are two storage options in the Session State Layer:

in the client side (Client Session State pattern), or on the server side (Server Session State pattern). In terms of storing sessions, we can design most web applications from merely stateless objects, and we can store session states only when we have to do so.

In case there is a force to have some remote objects in the system, there are two simple solutions offered in the Distributed Layer. The Remote Facade pattern acts as a facade for fine-grained objects that are placed on remote sites. The Data Transfer Object pattern acts as a partner for the Remote Facade pattern by bundling all the data that a client needs. Sometimes the Table Data Gateway pattern can return information from a query in the form of a Data Transfer Object.

There are patterns that do not belong to any of the above layers and can be considered as independent patterns. These patterns lie in the Base Layer. The Layer Supertype pattern acts as a supertype for all the objects in a layer. The Record Set pattern is an important pattern for representing tabular data as in-memory objects. Although most of the platforms offer a Record Set, the designers can create their own. The Money pattern is a very useful pattern when there is a need to work with different currencies and perform exchange conversions. The Gateway pattern acts as wrapper pattern that wraps the API code into a class which is similar to a regular object. The Mapper pattern acts as a mapping layer between two subsystems that need to stay ignorant of each other.

4.2 *PofEAA* Rules

As explained in Section 3.3, the most important behind-the-scene cornerstone of the PLV is a set of rules that drives the decision making engine of the main three modules: Pattern Structural Verifier (PSV), Pattern Language Syntactic Verifier (PTV), and Pattern Language Semantic Verifier (PMV). Corresponding to these modules, we need three groups of rules: **Structural, Syntactical**, and **Semantic**.

In Section 3.3, we explained how the rules must be extracted, categorized, and expressed in a formal way that is clear and precise for a programmer who is responsible to hard code those rules into the PLV modules. Extracting the rules that govern a PL, and classifying them into appropriate categories, is a difficult task, because these rules are often hidden between the lines of the texts that describe the patterns. This rule extraction and rule classification is a critical prerequisite step in building a PLV for a given PL, therefore, it should be done with enough care. In this section, we discuss how *PofEAA* rules are extracted, classified, and then formalized for the PLV process.

Patterns emerge from the experience of the experts [GHJV95, p. 1]. Hence, the *PofEAA* book, like many other pattern books, contains the advices for the designers, particularly for the novice designers. In a PL, the advices act as a structuring mechanism that lead a novice designer in selecting appropriate patterns one after another. This process continues until the whole system is designed. These advices are what we finally turn into the rules that are the basis for the PLV.

Regarding the selected patterns of *PofEAA*, we have extracted 74 advices from the book and bracketed them into three classes: *Structural, Syntactic*, and *Semantic.* Table 12 is an excerpt from these advices. The complete set of advices is displayed in Appendix A.2. Note that selecting the advice number (A#) and the advice classification (type) is our choice, but the descriptions are from the book. In the following sections, we refer to the advices in Table 12 by the advice number.

We try to preserve a one-to-one relationship between these advices and the formal rules that will be defined for the PLV for *PofEAA*. However, it is possible that one advice is the root for more than one rule, e.g., advice A47.

It should be noted that the advices extracted from the book reflect the author's (Martin Fowler) experience in working on enterprise applications. Some of the advices are not accurate enough, especially the syntactic and semantic ones. Most of the advices are about alternatives, and in some cases two advices may contradict each other. In case of any imprecision, ambiguity, or contradiction, the issue must be resolved in the course of formalizing the advice into a rule for the PLV. Resolving the issues is not an easy task, and needs expertise. Sometimes, one of the conflicting suggestions must be selected, and the others must be ignored. Sometimes, a vague suggestion needs interpretation. For instance, interpreting the word "usually" in advice A18: "The Table Data Gateway is usually stateless" is a subjective matter. Again, these issues must be resolved during the formalization of the advices into the rules.

In the following sections, we elaborate on how the advices of *PofEAA* are formalized into the rules. For each class of rules, we give some examples from Table 12, and using the formalisms proposed in Section 3.3, we obtain the corresponding formal rules. These formal rules will then be used to make the "PLV for *PofEAA*."

4.2.1 Structural Rules

Structural rules are those that describe the essence and the structure of an individual pattern. One important step in specifying the structure of a pattern is to select a "Sign"

Table 12: Advices from the *PofEAA* Book [Fow02]

A#	Type	Description (*PofEAA* book page#)
A04	Semantic	"If you have an environment like .NET or Visual Studio, then that makes a Table Module much more attractive." (p. 30)
A13	Syntactic	"A simple Domain Model can use Active Record, whereas a rich Domain Model requires Data Mapper." (p. 117)
A14	Syntactic	"A rich Domain Model is better for more complex logic, but is harder to map to the database." (p. 117)
A18	Structural	"A Table Data Gateway has a simple interface, usually consisting of several find methods to get data from the database and update, insert, and delete methods...The Table Data Gateway is usually stateless." (p. 144)
A23	Syntactic/ Semantic	"[for presentation layer] Your tooling may well make your choice for you. If you use Visual Studio, the easiest way to go is Page Controller and Template View. If you use Java, you have a choice of Web frameworks to consider. Popular at the moment is Struts, which will lead you to a Front Controller and a Template View." (p. 99)
A25	Structural	"A Front Controller handles all calls for a Web site, and is usually structured in two parts: a Web handler and a command hierarchy. The Web handler is the object that actually receives post or get requests from the Web server." (p. 344) "The Web handler is almost always implemented as a class rather than as a server page [...] The commands are also classes rather than server pages." (p. 345)
A29	Syntactic/ Semantic	"The essence of the choice between optimistic and pessimistic locks is the frequency and severity of conflicts." (p. 68) "Whereas Pessimistic Offline Lock assumes that the chance of session conflict is high and therefore limits the system's concurrency, Optimistic Offline Lock assumes that the chance of conflict is low." (p. 417)
A46	Semantic	"The parameter list of the insert method must be a subset of the parameter list of the update method." (p. 144)
A47	Syntactic/ Semantic	"You probably don't need a Service Layer if your application's business logic will only have one kind of client-say, a user interface-and its use case responses don't involve multiple transactional resources" (p. 137)
A49	Syntactic	"For this book I'm centering my discussion around an architecture of three primary layers: presentation, domain, and data source." (p. 19)
A51	Syntactic	"Often you'll find that there isn't quite a one-to-one relationship between Page Controllers and views." (p. 61)
A53	Syntactic	"Since it's a form of Mapper, Data Mapper itself is even unknown to the domain layer." (p. 165)

for each pattern. We select the names of the patterns that are written in bold in Table 11, as the "Sign" for the selected patterns of *PofEAA*.

The formalism that we defined for the structural rules (see Section 3.3.1) forces us to have clear and precise criteria written in English, so that an intermediate Object-Oriented (OO) programmer can interpret them in terms of programming language constructs. For the benefit of the programmer, a UML class diagram of the pattern may also be augmented to the criteria.

Amongst the extracted advices. 23 of them are structural advices. Each of these advices must be written in our proposed formalism, and if possible, supplemented by a UML class diagram.

For instance, to extract the structural rules for the Front Controller pattern, we investigate the description of the pattern, especially, "Intent," "Sketch," "How It Works," and "Examples" fields of the *pattern form.* The results of this investigation are presented in the advice A25 of Table 12, and in the "Intent" and the "Sketch" shown in Figure 19.

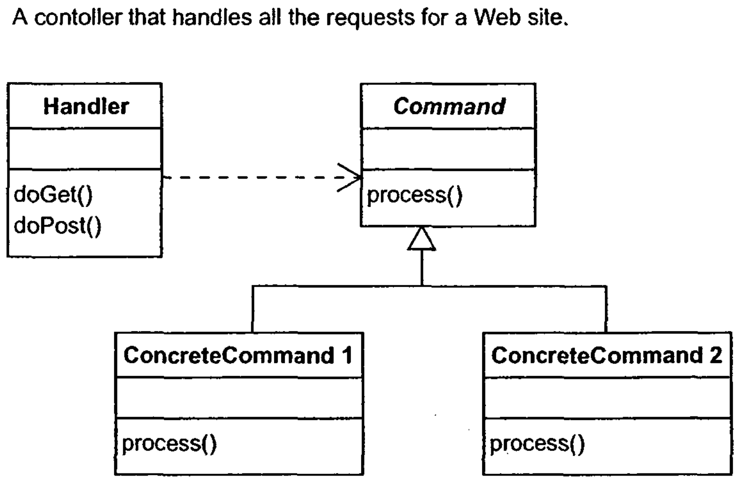


Figure 19: The Front Controller Pattern [Fow02, p. 344]

The information in advice A25 and Figure 19 is sufficient to define a set of eight criteria for the structural rules of this pattern, as shown in Figure 20. The rules are clear enough that there is no need to augment them with a class diagram.

1. There is a `Front Controller` (=`Handler`) class in the model.
2. There are at least two operations (`doGet` and `doPost`) in the Handler class.
3. The `Handler` class has a client dependency to a `Command` class.
4. The `Command` class is abstract.
5. The `Command` class has at least one `process` operation.
6. The `Command` class has at least one `Concrete Command` child class.
7. A `Concrete Command` class is concrete.
8. A `Concrete Command` class has at least one `process` operation.

Figure 20: *PofEAA* Rule Set - Part I: Structural Rules (A Sample Rule Showing the Structure of the Front Controller Pattern)

It is worth mentioning that these rules all together are considered as one structural rule. The same procedure is performed for all the 23 selected patterns, and the structural rules of all patterns are extracted. The result is called "*PofEAA* Rule Set - Part I: Structural Rules" and is shown in Appendix A.3.1.

4.2.2 Syntactic Rules

As it was discussed in Section 3.3.2, syntactic rules of PLV are divided into two groups. The first group shows the organization of patterns, i.e., which patterns are located in which layers. The second group, specifies the relationships and dependencies between patterns. In the following, for each group, first we show how the syntactic advices are extracted from the *PofEAA* book, then we discuss how those advices can be written as rules using the formalism defined in Section 3.3.2, and finally we show how the rules are inserted into the "*PofEAA* Rule Set."

Pattern-Layer Relationships The first group of syntactic rules, are derived from two sources: 1) The grouping of patterns into chapters in the *PofEAA* book, and 2) the explanations given in "Part 1" of the book about the optionality of some of the patterns or layers. We have already seen, in Figure 18, the first attempt in dividing 23 selected patterns into layers. Note that, there are two minor deviations between the layering of patterns in Figure 18 and what is proposed in the book. The first deviation is that we have separated the "Service Layer" pattern from the patterns of Domain Layer. The second deviation is that we have divided the patterns of the Presentation Layer into two sub-layers: the Controller Layer and the View Layer.

There are more details that are not represented in Figure 18, for example, the figure does not provide information about the mandatory or optional layers. Such information is extracted from "Part 1" of the book and is recorded as syntactic advices in Table 12.

For instance, advice A49 clarifies that there are three mandatory layers in the model. Also, advice A47 reveals that the "Service Layer" is not a mandatory layer and its existence depends upon the designer's choice.

We proposed a formalism for precisely presenting the layering of patterns of a PL (See Section 3.3.2). The following is the formal representation of the advices A49 and A47, supposing that all the layers lie in a root model named "pofeaa model."

$$\textit{pofeaa model} \supset \textit{presentation} \;.\; \textit{service}^{?(\text{Designer wants Service Layer})} \;.\; \textit{domain} \;.\; \textit{datasource}$$

In total, there are 16 advices about the organization of patterns. Investigating those advices along with Figure 18, and converting the advices into the formal rules, we obtained the "*PofEAA* Rule Set - Part II: Syntactic Rules (Pattern Organizations)." These rules are shown both in Figure 21 and Appendix A.3.2.

pofeaa model $\supset$ *main layer . auxiliary layer**
main layer $\supset$ *presentation . service*$^{?(C41)}$ *. domain . datasource*
presentation $\supset$ *controller . view*
auxiliary layer $\supset$ *base** , *distributed*$^{?(C42)}$, *concurrency*$^{?(C43)}$, *sessionstate*$^{?(C44)}$

controller $\ni$ *Page Controller , Front Controller*
view $\ni$ *Template View , Transform View*
service $\ni$ *Service Layer*
domain $\ni$ *Domain Model, Table Module, Transaction Script*
datasource $\ni$ *Data Mapper, Active Record, Table Data Gateway, Row Data Gateway*

base $\ni$ *Record Set, Layer Supertype, Money, Mapper, Gateway*
distributed $\ni$ *Remote Facade, Data Transfer Object*
concurrency $\ni$ *Optimistic Offline Lock, Pessimistic Offline Lock*
sessionstate $\ni$ *Client Session State, Server Session State*

C41: Designer wants Service Layer
C42: Designer wants Distributed Layer
C43: Designer wants Concurrency Layer
C44: Designer wants Session State Layer

Figure 21: *PofEAA* Rule Set - Part II: Syntactic Rules (Pattern Organizations)

Pattern-Pattern Relationships The second group of syntactic rules, that defines the relationship between patterns, can be extracted by investigating the pattern descriptions given by the *pattern form*, especially the fields: "Applicability" and "When to use it." Moreover, a "Cheat Sheet" is printed inside the back cover of the *PofEAA* book, which can also be considered a useful source for understanding the dependencies between patterns. This information is extracted and recorded as the advices.

We proposed a formalism for defining the relationships *uses*, *conflicts*, and *refines*, for precisely presenting the relationships between patterns of a PL (see Section 3.3.2).

As an example of a *uses* rule, consider advice A51 which says there are two alternative view patterns that can be used by a Front Controller pattern. Using the *alternative uses* formalism, we write the following rule.

$$\textit{Page Controller} \longrightarrow \textit{Template View} \mid \textit{Transform View}$$

As an example of a *conditional uses* rule, consider advice A23 which tells us how the selection of the tool will determine which view pattern should be used by a controller. Using the *conditional uses* formalism, we write the following rule.

$$\textit{Front Controller} \overset{\text{Tool=Java}}{\longrightarrow} \textit{Template View}$$

As an example of a *conflicts in layer* rule, consider advice A29 which says there is a choice between the Optimistic Offline Lock pattern and the Pessimistic Offline Lock pattern.

While it is not mentioned explicitly, we know that it is not possible to have both patterns for controlling the conflicts for the same unit of work. Hence, the resulted rule is:

Optimistic Offline Lock $\overset{concurrency}{\leftrightarrow}$ *Pessimistic Offline Lock* {Two patterns are applied for the same unit of work}

As an example of a *refines* rule, consider advice A53 which says the Data Mapper pattern is a special case of the Mapper pattern. The corresponding rule is as follows.

$$\textit{Data Mapper} \upharpoonleft \textit{Mapper}$$

There are 27 advices about the relationship between patterns. Using the formalism proposed in Section 3.3.2, we converted them into syntactic rules, and obtained the "*PofEAA* Rule Set - Part III: Syntactic Rules (Pattern Relationships)." These rules are shown both in Figure 22 and in Appendix A.3.3.

$\overline{\textit{Page Controller}} \longrightarrow \textit{Template View} \mid \textit{Transform View}$
$\overline{\textit{Front Controller}} \longrightarrow \textit{Template View} \mid \textit{Transform View}$
$\overline{\textit{Page Controller}} \xrightarrow{Tool=.NET} \textit{Template View}$
$\overline{\textit{Front Controller}} \xrightarrow{Tool=Java} \textit{Template View}$

$\textit{Template View} \xrightarrow{\text{C41}} \textit{Service Layer}$
$\textit{Transform View} \xrightarrow{\text{C41}} \textit{Service Layer}$
$\textit{Service Layer} \longrightarrow \textit{Domain Model} \mid \textit{Table Module}$

$\textit{Template View} \xrightarrow{\neg\text{C41}} \textit{Domain Model} \mid \textit{Table Module} \mid \textit{Transaction Script}$
$\textit{Transform View} \xrightarrow{\neg\text{C41}} \textit{Domain Model} \mid \textit{Table Module} \mid \textit{Transaction Script}$
$\textit{Page Controller} \xrightarrow{\neg\text{C41}} \textit{Domain Model} \mid \textit{Table Module} \mid \textit{Transaction Script}$
$\textit{Front Controller} \xrightarrow{\neg\text{C41}} \textit{Domain Model} \mid \textit{Table Module} \mid \textit{Transaction Script}$

$\textit{Domain Model} \xrightarrow{\text{C21}} \textit{Active Record}$
$\textit{Domain Model} \xrightarrow{\text{C23}} \textit{Data Mapper}$

$\textit{Table Module} \longrightarrow \textit{Table Data Gateway} \mid \textit{Row Data Gateway}$
$\textit{Transaction Script} \longrightarrow \textit{Table Data Gateway} \mid \textit{Row Data Gateway}$

$\textit{Table Data Gateway} \longrightarrow \textit{Record Set}\ \{\text{C111}\}$
$\textit{Table Data Gateway} \xrightarrow{C42} \textit{Data Transfer Object}$

$\textit{Data Mapper} \overset{datasource}{\longleftrightarrow} \textit{Active Record}$
$\textit{Table Data Gateway} \overset{datasource}{\longleftrightarrow} \textit{Row Data Gateway}\ \{\text{C112}\}$
$\textit{Optimistic Offline Lock} \overset{concurrency}{\longleftrightarrow} \textit{Pessimistic Offline Lock}\ \{\text{C112}\}$
$\textit{Client Session State} \overset{sessionstate}{\longleftrightarrow} \textit{Server Session State}\ \{\text{C112}\}$

$\textit{FrontController} \uparrow \textit{Controller}$
$\textit{PageController} \uparrow \textit{Controller}$
$\textit{Data Mapper} \uparrow \textit{Mapper}$
$\textit{Table Data Gateway} \uparrow \textit{Gateway}$
$\textit{Row Data Gateway} \uparrow \textit{Gateway}$

C21: Domain Structure is Simple
C22: Domain Structure is Moderate
C23: Domain Structure is Complex
C41: Designer wants Service Layer
C42: Designer wants Distributed Layer
C43: Designer wants Concurrency Layer
C44: Designer wants Session State Layer

C111: The return type of every `find()` operation in Table Data Gateway pattern is Record Set
C112: Two patterns are applied for the same unit of work

Figure 22: *PofEAA* Rule Set - Part III: Syntactic Rules (Pattern Relationships)

4.2.3 Semantic Rules

The semantic rules of PLV aim to catch two types of errors: 1) conflicts between the applied patterns and the context information, and 2) the inconsistencies between the features of applied patterns. Context information includes information about the environment of the system. Examples for context information are: the implementation tool, the designer's expertise, and the domain complexity.

For extracting semantic advices that govern the *PofEAA*, the pattern descriptions in the *pattern form* must be investigated carefully. The clue is to look for one of the concrete samples of the context information in the pattern description. Then the advices must be rewritten as the semantic rules using the formalism proposed in Section 3.3.3.

As an example of a semantic advice that checks the conflicts between the applied patterns and the context information, consider advice A14 about the effect of the domain complexity on the pattern used for the Domain Layer. Using the formalism proposed in Section 3.3.3, this advice is represented as the semantic rule:

$$\textit{Domain Model} \approx \{\text{Domain structure is complex }\}$$

As an example of a semantic advice that deals with the inconsistencies between the features of applied patterns, consider advice A46 about the correspondence of the parameters of the insert and update methods in the Table Data Gateway pattern. This advice is converted to the formal rule:

$$\textit{Table Data Gateway} \approx \{\textit{insert() parameter list} \subseteq \textit{update() parameter list}\}$$

Amongst the extracted advices, 17 of them are semantic advices. By interpreting those advices into the formal rules, we obtained the "*PofEAA* Rule Set - Part IV: Semantic Rules." These rules are shown both in Figure 23 and in Appendix A.3.4.

Page Controller $\approx$ {C11}
Front Controller $\approx$ {C12}
Template View $\approx$ {C61}
Transform View $\approx$ {C62}
Transaction Script $\approx$ {C11 and C21 and C31}
Table Data Gateway $\approx$ {*insert() parameter list* $\subseteq$ *update() parameter list*}
Active Record $\approx$ *Template View* {C121}

Service Layer $\approx$ {C41}
Remote Facade $\approx$ {C42}
Data Transfer Object $\approx$ {C42}
Optimistic Offline Lock $\approx$ {C43 and C51}
Pessimistic Offline Lock $\approx$ {C43 and C52}
Client Session State $\approx$ {C44}
Server Session State $\approx$ {C44}

C11: Tool is .Net
C12: Tool is Java

C21: Domain structure is simple
C22: Domain structure is moderate
C23: Domain structure is complex

C31: Designer is novice
C32: Designer is intermediate
C33: Designer is expert

C41: Designer wants Service Layer
C42: Designer wants Distributed Layer
C43: Designer wants Concurrency Layer
C44: Designer wants Session State Layer

C51: Chance of conflict is low
C52: Chance of conflict is high

C61: View is built using HTML
C62: View is built using XSLT

C121: The parameters of the operations of the Active Record pattern must match with the attributes of Template View

Figure 23: *PofEAA* Rule Set - Part IV: Semantic Rules

4.3 *PofEAA* UML Profile

In Section 3.5, we introduced PLV as a profile-driven process. That means, a prerequisite for having such a PLV for a PL, is to define a UML profile for the underlying PL. In the architecture of the PLV given in Figure 17, this profile is called Pattern Language UML Profile (PLP). This section is dedicated to explain how this profile, which we call it the "*PofEAA* UML Profile," is defined.

For defining a UML profile for *PofEAA* PL, we follow the profile definition approach introduced by Bran Selic [Sel07] (see Section 2.2.1). To summarize, Selic's approach for defining a UML profile for a language consists of two steps.

1. Define a domain model (metamodel) for the language.

2. Map the domain model onto the UML metamodel.

The next two sub-sections show how these two steps are taken for *PofEAA*. The remaining sub-sections elaborate on the stereotypes, the tagged values, and the constraints of the *PofEAA* UML Profile.

4.3.1 Defining the *PofEAA* metamodel

At the first step of Selic's approach, we need a domain model for our PL. This domain model is in fact the metamodel of the language. The metamodel should consist of the fundamental concepts of the domain, their relationships, the constraints on these concepts, the notation, and the semantics of the language.

Our work in Section 4.2 makes this step easier. For our selected patterns from *PofEAA*, Figure 18 plays the role of the domain model, because it displays the concepts of the domain. In addition to Figure 18, part I and part II of the "*PofEAA* Rule Set" (see Figure 20 and Figure 21) must be taken into consideration, to discover more concepts of the domain. For the relationships between the concepts, we can utilize part III of the "*PofEAA* Rule Set" (see Figure 22). The semantics of the language is what we have seen in part IV of the "*PofEAA* Rule Set" (see Figure 23). The constraints on the concepts are the ones that are mentioned in different parts of the "*PofEAA* Rule Set." For the notation of our language, we use both UML-ish diagrams (Figure 18 is a UML package diagram) and the formalisms introduced in Section 3.3.

4.3.2 Mapping *PofEAA* metamodel to UML metamodel

In the second step of Selic's approach, we should map each of the concepts of the domain model of our language into one of the UML metamodel classes. At this step, we investigate which concepts of the UML metamodel need to be extended to fulfill the requirements of our language concepts. For those concepts, we define *stereotypes.* We should be careful not to have conflicts between our concepts and the base meta-classes in UML.

Before showing how this step can be applied for *PofEAA*, we need to discuss an extra step which we think is mandatory when the concepts in the language's domain model are compound. By a compound concept, we mean a concept which is constructed from several single (atomic) concepts. We call this extra step "decomposition."

In the pattern language world, in which each pattern is considered as one concept, many of the concepts are compound. In other words, a pattern has a structure (typically represented by a UML class diagram) and consists of several other concepts (such as classes). The decomposition step aims to find the atomic concepts that could be matched to the UML meta-classes clearly.

For instance, the Front Controller pattern, shown in Figure 19, is a concept in the domain model of *PofEAA* PL. This concept is a compound concept, consisting of three atomic concepts (Handler, Command, and ConcreteCommand classes). Note that we use "Front Controller" as the sign of this pattern, hence, instead of Handler, we make use of the name FrontController.

In addition to the Front Controller, there are four other compound concepts in the selected patterns from *PofEAA*: the Record Set pattern which its decomposition results in adding concepts Table, Row, and Column; the Row Data Gateway pattern which needs a Finder class; the Remote Facade pattern which needs a class as the owner of bulk-accessor methods; and the Money pattern which needs a Currency class as the type of its currency field.

When the decompositions are done, i.e., each compound concept in the domain model is replaced by its constituent elements, then performing step two of the Selic's approach is possible. Figure 24 represents the result of applying this step for our selected patterns of *PofEAA*. It shows how the metamodel (domain model) of the *PofEAA* PL is mapped into the UML metamodel. The figure indicates that the concepts of our domain model are mapped as extensions of four UML meta-classes: package, class, operation, and attribute. Note that, for the sake of simplicity, the operations and attributes of the classes were not shown in Figure 18.

In Figure 24, the gray boxes and their associations are copied from the UML metamodel [Obj05b] for clarification, i.e., the gray boxes are the UML meta-classes. The white boxes are the concepts of the domain model of our language (see Figure 18). An arrow (↑) from a language concept (white box) to a UML meta-class (gray box) should be interpreted as an *extension*. That means, each stereotype *extends* one of the meta-classes of the UML. Obviously, the white boxes show the stereotypes of the "*PofEAA* UML Profile." The next section explains these stereotypes in more details.

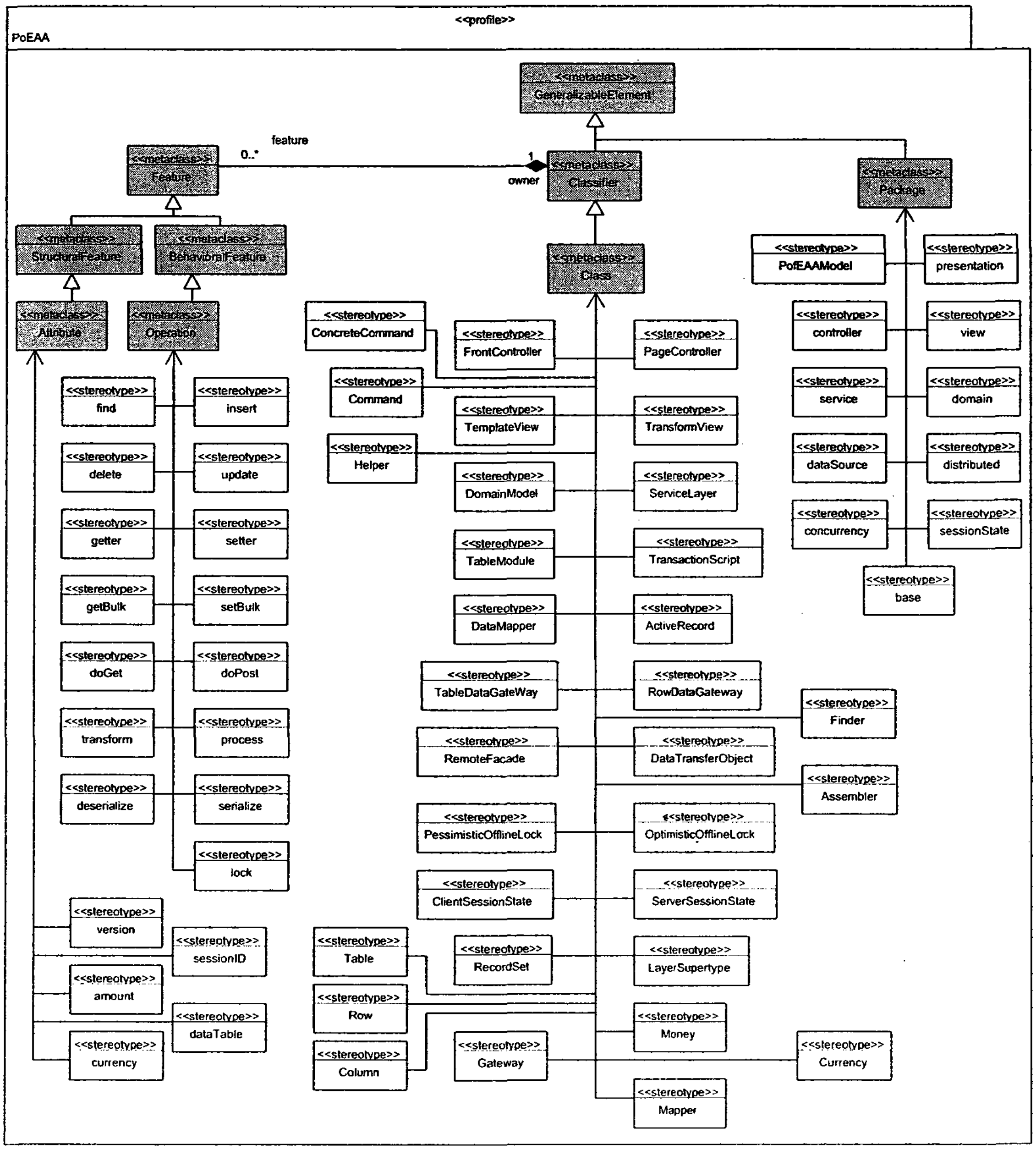


Figure 24: Mapping the *PofEAA* metamodel into the UML metamodel

4.3.3 Stereotypes of the *PofEAA* Profile

As can be seen in Figure 24, in "*PofEAA* UML Profile," we have four groups of stereotypes: package-based, class-based, operation-based, and attribute-based. The first group, package-based stereotypes, contains 11 stereotypes. One stereotype named «PofEAAModel» is considered for the whole model. It is supposed that this is the root package of the design, i.e., this package includes all other packages. Three stereotypes are defined corresponding to the three main layers: «presentation», «domain», and «dataSource». Two stereotypes are defined for the sub-layers of the presentation layer: «controller» and «view». Five stereotypes, «service», «distributed», «concurrency», «sessionState», and «base» are considered for the supporting layers. Therefore, we have defined 11 stereotypes that extend the meta-class "Package" of the UML metamodel. Figure 25 shows the packages in our profile.

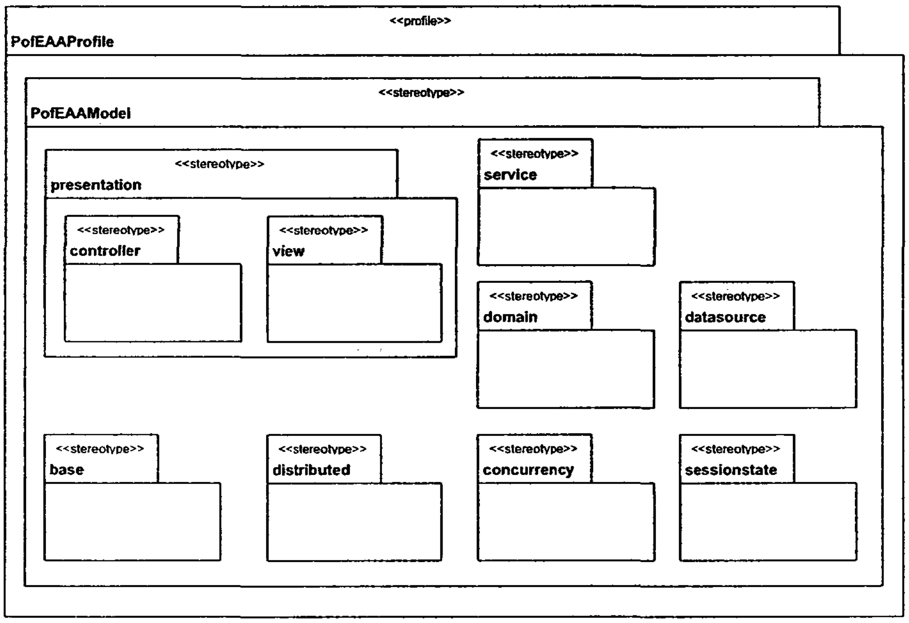


Figure 25: The Packages in the *PofEAA* UML Profile

The second group, class-based stereotypes, has 32 elements. There are 23 stereotypes that are named after the 23 selected patterns of *PofEAA*. As it was discussed in Chapter 3, we call these stereotypes the "Signs" of the patterns. For each pattern there is a unique "Sign" stereotype, which is considered by the PSV, as the starting point for checking the structure of the pattern. The remaining nine stereotypes are defined for the classes that are found after decomposition of compound patterns. «Command» and «ConcreteCommand» are defined for the Front Controller pattern. «Helper» is defined for the Template View pattern. «Finder» is defined for the Row Data Gateway pattern. «Table», «Row», and

«Column» are defined for the Record Set; «Assembler» is defined for the Data transfer Object, and «Currency» is defined to specify the type for "currency" field of the Money pattern.

The third group, operation-based stereotypes, includes 15 stereotypes that extend the meta-class "Operation" of the UML metamodel. These stereotypes are used when there are mandatory operations for a pattern. Stereotypes «find», «insert», «delete», and «update», are required for patterns that need CRUD (Create, Read, Update, and Delete) operations. Stereotypes «getter» and «setter», can be used generally to specify the accessor methods of a class, but particularly these stereotypes along with «getBulk» and «setBulk» are defined for the RemoteFacade pattern. Stereotypes «doGet», «doPost», and «process» are used by the Controller patterns. Stereotype «transform» indicates the transformer operation in the Transform view pattern. Stereotypes «serialize» and «deserialize» are defined for the Data Transfer Object pattern. Stereotypes «lock» is defined for the Offline Concurrency patterns.

The fourth group, attribute-based stereotypes, includes 5 stereotypes that extend the meta-class "Attribute" of the UML metamodel. Stereotype «version» is required when the Optimistic Offline Lock pattern is applied. Stereotypes «sessionID» is required for keeping the ID of each session in Session State patterns. Stereotypes «amount» and «currency» are defined to specify the fields of the Money pattern. The «dataTable» stereotype is used by the Table Module pattern to indicate the attribute which contains the name of the Data Base (DB) table.

For instance, pattern Front Controller (see Figure 19) involves three classes and three operations. Therefore, for detecting this pattern we need six stereotypes: «FrontController», «Command», and «ConcreteCommand» as class-based stereotypes, and «doPost», «doGet», and «process» as operation-based stereotypes. These stereotypes (particularly the first one), when applied, show the designer's intention for using the Front Controller pattern. Note that the Sign («FrontController») must be applied on the Handler class.

4.3.4 Tagged Values of the *PofEAA* Profile

Tagged values are used to attach additional meta-attributes to a stereotype in order to access information about the model, such as the context information or the configuration management properties. It is worth noting that a tagged value is not the same as an attribute of a class. In fact, a tagged value is meta-data and its value applies only to the related element and not to the instance.

In *PofEAA* profile, we have defined nine tagged values, all applicable on stereotype «PofEAAModel», as slots to capture information about the model, the designer, and the development environment. The tagged values are introduced in Table 13, which defines the tag, type of each tag, a multiplicity indicating how many individual values can be assigned to it, and alternative values for the tag. A lower bound of zero for multiplicity implies that the tagged value is optional.

Table 13: Tagged Values for stereotype «PofEAAModel» in *PofEAA* UML Profile

Tag	**Type**	**Mult.**	**Values**
ServiceLayer	String	[0..1]	Yes , No
DistributedLayer	String	[0..1]	Yes , No
ConcurrencyLayer	String	[0..1]	Yes , No
SessionStateLayer	String	[0..1]	Yes , No
ChanceOfConflict	String	[0..1]	Low , High
ViewBuilt	String	[0..1]	HTML , XSLT
Tool	String	[0..1]	Java , .Net
Complexity	String	[0..1]	Simple , Moderate, Complex
Expertise	String	[0..1]	Novice , Intermediate , Expert

The tags "ServiceLayer," "DistributedLayer," "ConcurrencyLayer," and "SessionState-Layer" indicate whether or not the designer decides to have the corresponding layer in his/her design. The values are simply "Yes" or "No" strings.

The tag "ChanceOfConflict" determines which one of the concurrency patterns (Optimistic Offline Lock or Pessimistic Offline Lock) is appropriate for the current design. The value "High" means the possibility of transactional conflict in the system is high, therefore, the Pessimistic Offline Lock pattern is preferred. Similarly, the value "Low" means the Optimistic Offline Lock pattern is more appropriate.

The tag "ViewBuilt" specifies how the view of the presentation is built, hence, the value of this tag discriminates the pattern that is used for the View. The value "HTML" leads to the Template View pattern, while the value "XSLT" (Extensible Stylesheet Language Transformations) encourages the usage of the Transform View pattern.

The tag "Tool" is defined for information about the implementation environment. The alternative values are the name of the platform which is used for developing the system, e.g., "Java" or ".Net." This tag is used in constraints that check the compatibility of a pattern with the development tools. For instance, when the value is "Java," a stereotype «TableModule» in the Domain Layer will trigger a semantic error, since the Table Module pattern is better matched with the ".Net" platform.

In order to check the complexity of the domain model, and then to verify which pattern must be applied in the Data Source Layer, we have defined a tag named "Complexity."

There are three possible values for this tag, "Simple," "Moderate," or "Complex." For instance, when the value is "Simple," the objects identified by the Domain Model pattern can be integrated with an Active Record pattern to have access to the data base, however, when the value is "Complex," the Domain Model objects must use a Data Mapper pattern.

The tag "Expertise" reflects the level of experience of the designer, and its value affects the choice of patterns in the Domain Layer. The values are one of the three strings: "Novice," "Intermediate," or "Expert." For example, for novice designers, applying the Domain Model pattern is discouraged, but for expert designers it is encouraged.

4.3.5 Constraints of the *PofEAA* Profile

In addition to the stereotypes and tagged values, a UML profile may contain several constraints. Constraints are invariants that can be attached to every model element, including the stereotypes. When a constraint is defined for a stereotype, applying that stereotype on a model element causes the constraint to be checked. There are two approaches for specifying a constraint: formally using the OCL language, or informally using a natural language. It is obvious that to have automatic constraint checking, the constraints should be written in OCL and the tool should have support for profile (and OCL) checking.

In Section 3.5, we discussed the pros and cons of the above two approaches. We justified our decision of performing a two-step procedure: first, representing the constraints using the formalism which is defined in Section 3.3, and second, hard coding the constraints into a modeling tool using a programming language. Our formalism is a grammar-like notation, in which, rules can be augmented with textual comments or conditions. The first step is already taken, since the "*PofEAA* Rule Set" that we defined in Section 4.2 is indeed the constraints of "*PofEAA* UML Profile." As the second step, to complete the definition of our profile, these constraints must be hard coded as the modules of a PLV for *PofEAA*. This is discussed in the next section.

The equivalence of the "PofEAA Rule Set" and the constraints of "*PofEAA* Profile," requires some clarification. In the former, we divided the rules into three parts: structural, syntactic, and semantic, while in the latter, the constraints are typically defined for the stereotypes. Specifying which rules are related to which stereotype is not a difficult task. As a rule of thumb, we can say that in the "PofEAA Rule Set," each rule is related to the name of the pattern or layer which appears on the left-hand side of the rule. For instance, "*Front Controller* $\xrightarrow{\text{Tool=Java}}$ *Template View*" should be considered as a constraint for the stereotype «FrontController».

To wrap up the profile discussion, Table 14 shows some statistics about the "*PofEAA* UML Profile," and Table 15 shows some of its stereotypes along with related constraints and tagged values. For some of the stereotypes, the constraints are given both in natural language and in OCL.

Table 14: Statistics about the *PofEAA* UML Profile

Stereotypes	**Tagged Values**	**Constraints**
63 (11 Package-based, 32 Class-based, 15 Operation-based, 5 Attribute-based)	9	70+

Table 15: Some Stereotypes of the *PofEAA* Profile

Name	«PofEAAModel»
Base Class	Package
Description	The root of the model
Tagged Values	ServiceLayer, DistributedLayer, ConcurrencyLayer, SessionStateLayer, ChanceOfConflict, ViewBuilt, Tool, Complexity, Expertise
Constraints	It should have at least three sub packages corresponding to three main layers of *PofEAA*. It might have five supplementary packages, subject to designer's decision.
Constraints in OCL	self.ownedElement → includes (p1, p2, p3:Package— p1.stereotype ='presentation' and p2.stereotype ='domain' and p3.stereotype ='datasource') and self.ownedElement → includes (p4:Package — p4.stereotype ='service' and self.getValue('ServiceLayer') = 'Yes')) and ...
Name	«presentation»
Base Class	Package
Description	The presentation layer package.
Constraints	It should have controller and view sub-packages.
Constraints in OCL	self.ownedElement → includes (p:Package— stereotype='controller') and self.ownedElement → includes (p:Package— stereotype='view')
Name	«domain»
Base Class	Package
Description	The domain layer package.
Tagged Values	complexity
Constraints	It should have patterns as the domain model of the system which are compatible with the context information, e.g., complexity of the domain model, the tool, and the expertise of the developers.
Name	«dataSource»
Base Class	Package
Description	The data source layer package.
Constraints	It should have patterns for connecting to the database which are compatible with the patterns in the domain model.
Name	«TableDataGateway»
Base Class	Class
Description	An object that acts as a Gateway to a database table. One instance handles all the rows in the table.
Constraints	It should have find, insert, delete and update operations. The return type of find operations should have stereotype «recordSet».
Constraints in OCL	self.BehavioralFeature → exists (o:Operation — name='find') and ...

4.4 ArgoPLV: A PLV for *PofEAA*

In this section, we use the UML profile defined for *PofEAA* in Section 4.3, to show how the PLP is implemented, and how four modules, PSV, PTV, PMV, and PLA, are hard coded into a modeling tool to build "A PLV for *PofEAA*." As the modeling tool, we have selected ArgoUML, hence, the resulted tool is called "ArgoPLV."

4.4.1 ArgoUML

ArgoUML [Tig09a] is an open-source UML modeling tool. The core ideas of ArgoUML are the result of Jason Robbins's PhD thesis [Rob99] titled "Cognitive Support Features for Software Development Tools." In February 1999, ArgoUML was made into an Open Source project.

ArgoUML has always been under development, and a dynamic development community is working on fixing the reported bugs as well as adding new features. The current version, as of date (March 1, 2009), is ArgoUML 0.26.2 which is more stable and has many more features than the original version. ArgoUML is written in Java and is available in three different formats: Java Web Start, installable, and source code. Current version of ArgoUML is based upon the NetBeans MDR [Mic09] implementation of UML metamodel which supports UML 1.4. For OCL, ArgoUML uses Dresden OCL toolkit [The09]. ArgoUML uses the Graph Editing Framework [Tig09b] (GEF, not to be confused with the Eclipse Graphical Editing Framework (GEF)) to edit UML diagrams. We downloaded the ArgoUML 0.26.2 source code and built it in Eclipse 3.3 [Fou09b].

ArgoUML is a UML modeling tool that supports all standard UML 1.4 diagrams: Use Case, Class, Sequence, Collaboration, State chart, Activity, and Deployment (includes Object and Component). There is no immediate plan to support UML 2.0 in ArgoUML. Besides features such as diagram editor and reverse engineering of compiled Java code, ArgoUML is a design critiquing system. As the creator of ArgoUML defines "A *design critic* is an intelligent user interface mechanism embedded in a design tool that analyzes a design in the context of decision-making and provides feedback to help the designer improve the design" [Rob99].

ArgoUML's main window has a toolbar, menu bar and four main panes: 1) Explorer, 2) Editing, 3) ToDo, and 4) Details. Figure 26 shows a snapshot of the main window of ArgoUML with four main panes specified. Explorer pane shows a hierarchical view of the current project file. Editing pane is an editor for the selected diagram of the model, e.g., the class diagram. ToDo pane contains the designer's ToDo List. Details pane shows the details of the selected object in the diagram or the selected ToDo Item from the ToDo List.

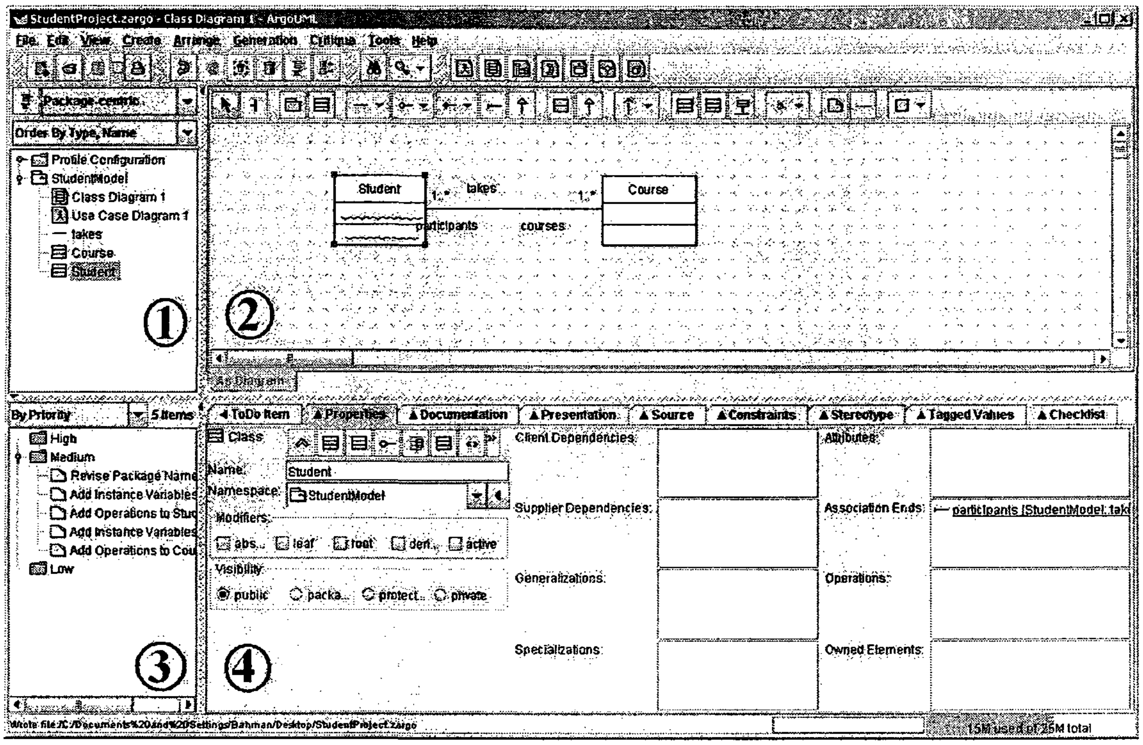


Figure 26: ArgoUML's window has four main panes: 1) Explorer, 2) Editing, 3) ToDo, and 4) Details.

ArgoUML's critiquing system is based upon a conceptual critiquing process called ADAIR (Activate, Detect, Advise, Improve, Record) [RR98]. Simply put, ArgoUML has predefined agents, called *critics*, that are constantly investigating the current model and if the conditions for triggering a *critic* hold, the *critic* will generate a ToDo Item (this item is called a *critique*) in the ToDo List. A ToDo Item contains a short description of the problem, some guidelines about how to solve the problem, and if there exists, a *wizard* which helps the designer solve the problem automatically. A ToDo Item generated by a *critic* will remain in the ToDo List until the origin of the problem is vanished, either manually by the designer, or by following the *wizards* proposed by the tool.

The *critics* run as asynchronous processes in parallel with the main ArgoUML tool. The *critics* are not intrusive, since the user can completely ignore them or disable one or all of them through the Critique menu. *Critics* and *wizards* are not user defined, since they all are written in Java and are compiled as part of the tool.

4.4.2 ArgoPLV Architecture

Figure 27 shows the architecture of ArgoPLV as an extension to ArgoUML. In the core of ArgoUML, the model is accessed via `org.argouml.model.Facade`, which is the facade object for the Model subsystem. In the Model subsystem, a set of Factories and Helpers are defined to allow the manipulation of the objects of the model. ArgoPLV Plugin is the result of several extensions to the ArgoUML architecture.

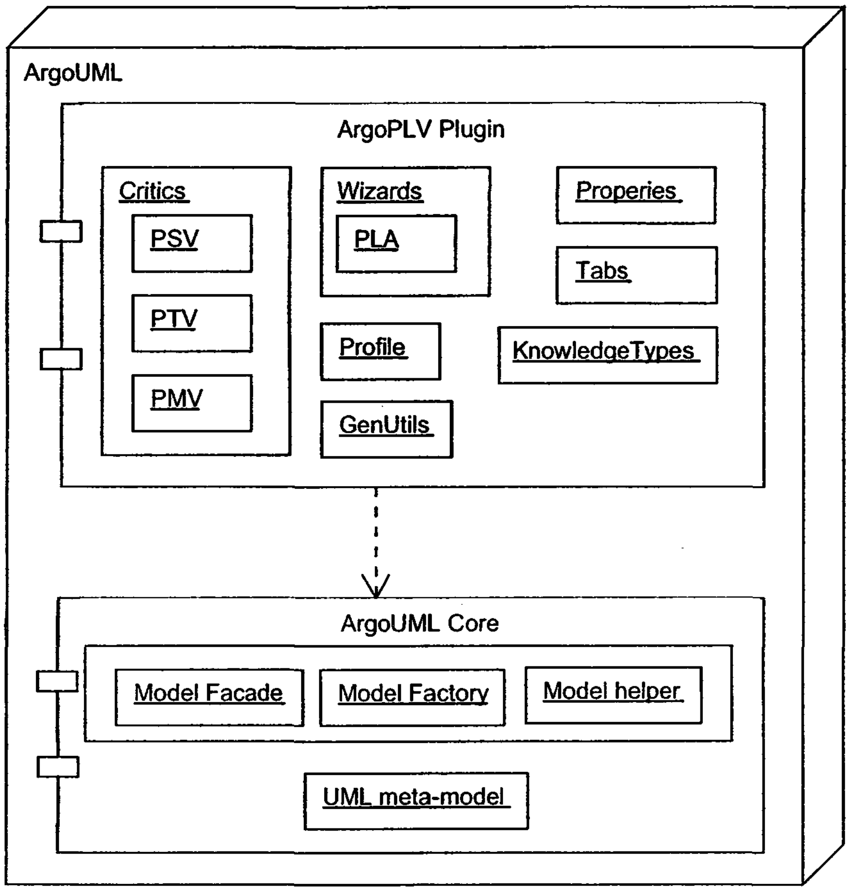


Figure 27: ArgoPLV Architecture

Three PLV modules (PSV, PTV, and PMV) are packaged into ArgoUML design *critics*. Each of the ArgoPLV *critics* is implemented as a class inherited from the following class: `org.argouml.uml.cognitive.critics.CrUML`. Each *critic* is registered with the class `org.argouml.cognitive.Agency`, then a designer thread is started to check whether the *critic* can find a problem in the current model. If a problem is found, a ToDo Item (critique) will be posted to the ToDo List. The fourth module (PLA), is packaged into the *wizards*. Furthermore, this module requires the user interface of ArgoUML to be extended by adding new Tabs to the Details Pane, and new categories of Knowledge Type to be added to the ToDo List. The technical details on building PLV modules are explained in the following.

4.4.3 PLP in ArgoPLV

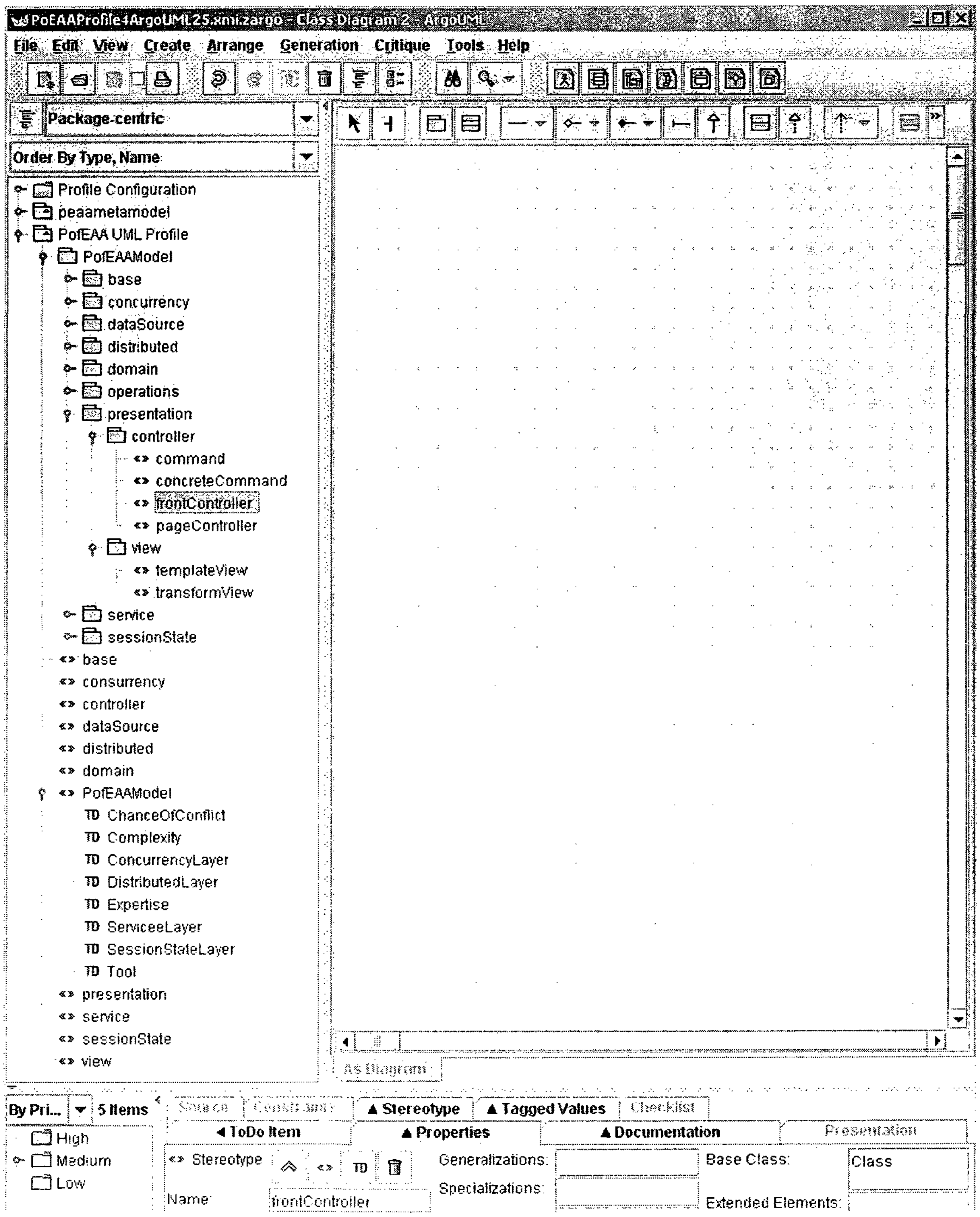
In Section 4.3, we introduced the stereotypes, the tagged values, and the constraints of the "*PofEAA* UML Profile". In this section, we explain how this profile is implemented in ArgoUML, to play the role of PLP in ArgoPLV.

Defining the stereotypes and tagged values of PLP in the ArgoUML tool is not a difficult task, however, the support is not straightforward. One approach for defining a profile in ArgoUML, is to create a dummy model, then define all the required stereotypes and tagged values in that model, and finally, export the model to an XMI file. This file can then be considered as a profile, to be loaded and applied on another model.

The constraints are not codified as part of the PLP due to the following reasons. First, ArgoUML does not have support for writing constraints at the metamodel level (note that our constraints are all at the metamodel level). Second, we have already explained (see Section 4.3.5) that our constraints (rules) are not completely written in OCL, instead, they are written using our defined formalism, enriched with class diagrams and informal comments written in English. Therefore, we decided to code the constraints of the PLP in Java inside the ArgoUML *critics*.

After the profile (stereotypes and tagged values) is defined, it can be applied on a model. Applying a profile is recently added as a feature to ArgoUML. In the ArgoUML versions 0.25 or higher, a new feature called "Profile Configuration" is added that allows the designer to load an existing profile (which is serialized in XMI) and apply it to the current model. By applying the profile, all the stereotypes and tagged values are available for using in the current model.

Figure 28 shows a snapshot of ArgoUML where the stereotypes and the tagged values of "*PofEAA* UML Profile" are defined. In the figure, "*PofEAA* UML Profile" is the name of the model (profile). The packaging of the stereotypes is only done for the sake of aesthetic reasons, e.g., the stereotypes «Command», «ConcreteCommand», «FrontController», and «PageController» are placed inside a package named "controller" which is inside a "presentation" package. which is inside a root "PofEAAModel" package. The stereotypes for the layers of the system are placed inside the root "PofEAAModel" package. The tagged values are defined for the stereotype «PofEAAModel».



```
Profile Configuration
peaametamodel
PofEAA UML Profile
  PofEAAModel
    base
    concurrency
    dataSource
    distributed
    domain
    operations
    presentation
      controller
        «» command
        «» concreteCommand
        «» frontController
        «» pageController
      view
        «» templateView
        «» transformView
    service
    sessionState
  «» base
  «» consurrency
  «» controller
  «» dataSource
  «» distributed
  «» domain
  «» PofEAAModel
    TD ChanceOfConflict
    TD Complexity
    TD ConcurrencyLayer
    TD DistributedLayer
    TD Expertise
    TD ServiceeLayer
    TD SessionStateLayer
    TD Tool
  «» presentation
  «» service
  «» sessionState
  «» view
```

```
High
Medium
Low
```

Stereotype		Generalizations:		Base Class:	Class
Name:	frontController	Specializations:		Extended Elements:	

Figure 28: Defining Stereotypes and Tagged Values of *PofEAA* Profile in ArgoUML

4.4.4 PSV in ArgoPLV

In ArgoPLV, the PSV is built by hard coding the structural rules of the *PofEAA* PL into the *critics*. In Section 4.2.1 we called these rules "*PofEAA* Rule Set - Part I: Structural Rules." Based on the critiquing idea of ArgoUML, for each kind of problem there should exist a *critic* class. Therefore, for each one of the *PofEAA* patterns, we have one *critic* which verifies the structure of that pattern and detects the errors. Each *critic* is indeed the hard coding of the structural rules for the corresponding pattern. Hence, PSV is implemented by 23 *critics* (one *critic* per pattern).

To see an example of how the PSV is coded into the *critics*, consider the structural rules (Criteria) of the Front Controller pattern, which is shown in Figure 20 as one sample rule from the "*PofEAA* Rule Set - Part I: Structural Rules." Based on these criteria, PSV starts by finding the Handler class, a class with stereotype «FrontController». Then it looks for two operations with stereotypes «doGet» and «doPost» in that class. The Handler class shall be a client of a Command class, a class with stereotype «Command». The Command class must be abstract and have an operation with stereotype «process». The Command class must also have at least one child. All children of the Command class shall be concrete classes with stereotype «ConcreteCommand». Each ConcreteCommand class in turn shall have an operation with stereotype «process».

The *critic* class which verifies the structure of the Front Controller pattern is class `CrFrontController`. An excerpt (the "predicate" method) from this *critic* is shown in two parts in Figure 29 and Figure 30. The "predicate" method is the heart of each *critic* which checks the conditions to see whether or not the *critic* must be triggered. The whole code of this *critic* is attached in Appendix A.5.2. Each *critic* is a thread which is running all the time and investigates every object in the model.

In line 3, by checking the current model element (called "dm"), we make sure that this *critic* deals only with the classes. The model elements are accessed via the facade object "org.argouml.model.ModelFacade." In line 7, we check the "Sign" of the pattern, and if it is not «FrontController», the *critic* returns without reporting any problem. The remaining lines of the code (in Figure 29 and Figure 30), check the Criteria of the pattern. This code, as well as other *critics*, uses the services of a General Utility class (`GU`) which is a singleton class for performing tasks such as finding a specific stereotype of an object, or finding a specific operation in a class. The Javadoc of the `GU` class is attached in Appendix A.5.1.

In case of any error, i.e., reaching any of the "return PROBLEM_FOUND" statements in the code, PLA is invoked to give an error message to the designer, and to guide him/her

```
public boolean predicate(Object dm, Designer dsgr) {
  if (dm == null) return NO_PROBLEM;
  if (! Model.getFacade().isAClass(dm)) return NO_PROBLEM;
  Object aClass = dm;
  // aClass should have stereotype <<FrontController>>, this is the sign
  // of pattern that is applied on the Handler class
  if (! GU.objectHasSte(aClass, "FrontController"))    /*SIGN*/
     return NO_PROBLEM;
  // Both doGet and doPost ops are required
  if (! GU.classHasSteOp(aClass,"doGet" ) ) return PROBLEM_FOUND;
  if (! GU.classHasSteOp(aClass,"doPost") ) return PROBLEM_FOUND;
  // Check if there is a client
  Collection depSet = Model.getFacade().getClientDependencies(aClass);
  if ( depSet.isEmpty() ) return PROBLEM_FOUND;
```

Figure 29: Predicate Method of the `CrFrontController` Critic (Part 1)

in fixing the problem. The PLA is introduced in Section 4.4.7.

If all the structural criteria of a pattern hold, a line will be added to the PIT, and the detected pattern is reported to the designer. While checking the structure of a pattern, the pattern elements are inserted into a list named "classNames" (see lines 48, 52, and 56 in Figure 30). This list is indeed the PIT.

4.4.5 PTV in ArgoPLV

PTV in ArgoPLV is built by hard coding the syntactic rules of the *PofEAA* PL into the *critics*. In Section 4.2.2, two groups of syntactic rules were defined for *PofEAA*:

1. Rules that check the organization of patterns: These rules are named "*PofEAA* Rule Set - Part II: Syntactic Rules (Pattern Organization)," and are represented in Figure 21.

2. Rules that check the relationship between patterns: These rules are named "*PofEAA* Rule Set - Part III: Syntactic Rules (Pattern Relationships)," and are represented in Figure 22.

In the following, we will elaborate how these two groups of rules are hard coded into the PTV module.

Rules that Verify the Pattern-Layer Relationships For checking the pattern organizations, two *critics* are written: `CrLayers` and `CrPatterns`. These *critics* are verified

```
 // at least one of the suppliers should have the COMMAND structure
 // An ABSTRACT class with stereotype <<Command>> and  one <<process>>
 // operation as well as at least one child with stereotype
 // <<ConcreteCommand>> and with one <<process>> operation
 boolean supplierFound = false;
 Iterator deps = depSet.iterator();
 while ( deps.hasNext() ) {
   Object dep = deps.next();
   Collection supplierSet = Model.getFacade().getSuppliers(dep);
   if ( supplierSet.isEmpty() ) continue;
   Iterator suppliers = supplierSet.iterator();
   while ( suppliers.hasNext() && !supplierFound) {
     // This should be the Command class
     Object supplier = suppliers.next();
     if ( GU.objectHasSte(supplier, "Command") ) {
       if (Model.getFacade().isAbstract(supplier)) {
         if (GU.classHasSteOp(supplier,"process")) {
           // We need at least one child which is concrete
           // and has process operation
           Collection children = Model.getFacade().getChildren(supplier);
           if ((children.isEmpty())) return PROBLEM_FOUND;
           Iterator child = children.iterator();
           while (child.hasNext()) {
             Object conCommand = child.next();
             // concrete command must be a class
             if (! Model.getFacade().isAClass(conCommand)) continue;
             //  concrete command class  must be concrete
             if (Model.getFacade().isAbstract(conCommand)) continue;
             if (!GU.objectHasSte(conCommand,"ConcreteCommand"))
               return PROBLEM_FOUND;
             if (!GU.classHasSteOp(conCommand,"process"))
               return PROBLEM_FOUND;
             // Now, report the correct usage of FC pattern
             classNames.add(Model.getFacade().getName(conCommand)+"
             -> Concrete Command");
           }
           supplierFound = true;
           classNames.add(Model.getFacade().getName(supplier)+"-> Command");
 } } } } }
 if ( ! supplierFound ) return PROBLEM_FOUND;
 PATTERN_FOUND = true;
 classNames.add(Model.getFacade().getName(aClass)+" -> Handler");
 patternLayer =
        Model.getFacade().getName(Model.getFacade().getNamespace(aClass));
 return NO_PROBLEM;
}
```

Figure 30: Predicate Method of the CrFrontController Critic (Part 2)

against the whole model (the package with stereotype «PofEAAModel»), since they are more general to be checked for a specific class.

The first *critic* class, CrLayers, checks the model to see if any of the mandatory or supplementary layers is missing. Note that the existence of a supplementary layer is subject to the designer's choice, by setting the corresponding Tagged Value. In terms of the *PofEAA* rules, this *critic* applies the the first four rules of the "*PofEAA* Rule Set - Part II," given in Figure 21. In case of any error, i.e., finding a missing layer, the PLA is called to report the error, and to help the designer in adding the missing layer.

The source code of the `CrLayers` class is attached in Appendix A.5.3. An excerpt from the code is shown in Figure 31. In line 2, an Iterator is defined to traverse on all the elements inside the root model. In line 6, we only consider the elements that are packages. Line 7, using a utility method (`hasStr()`), looks for the "presentation" layer. Line 11, looks for the "service" layer. In line 18, we check for errors. The non-existence of the "presentation" layer, or, the non-existence of the "service" layer while the designer has indicated that he/she wants this layer (i.e., utility method `needsServiceLayer()` returns "true"), are considered as errors.

Note that, if the Service Layer is found, but the designer has not requested it (i.e., utility method `needsServiceLayer()` returns "false"), this case is not considered as a syntactic error. This is indeed a semantic error which will be caught by the PMV module as it will be discussed in the next section (see Section 4.4.6).

```
1 //Lines Deleted. aPackage is the root PofEAA package.
2 Iterator innerElms = Model.getFacade().getOwnedElements(aPackage).iterator();
3 while (innerElms.hasNext()) {
4    Object elmnt = innerElms.next();
5    if ( elmnt != null ) {
6        if (Model.getFacade().isAPackage(elmnt)) {
7            if (GU.hasStr(elmnt, "presentation")) {
8               presentationFound = true;
9               prs = Model.getFacade().getName(elmnt);
10           }
11           else if (GU.hasStr(elmnt, "service")) {
12               serviceFound = true;
13               srv = Model.getFacade().getName(elmnt);
14           }
15  //Lines Deleted
16 }   }   }
17 //Lines Deleted
18 if ( !presentationFound || (!serviceFound && GU.needsServiceLayer())
19 //Lines Deleted
20    )
21   return PROBLEM_FOUND;
```

Figure 31: An Excerpt from the Source Code of class `CrLayers`

Remember that, the designer decides about having an optional (supplementary) layer by setting the value of the corresponding tagged value. For instance, if the designer intends to have a Service Layer in the model, he/she sets {ServiceLayer=Yes} for the root package of the design. To access the tagged values that capture the context information, the `CrLayers` class uses the utility methods from the "GU" class. For example, one of these utility methods (`needsServiceLayer()`) that determines whether or not the designer wants the Service Layer in the model, is shown in Figure 32. Note that, in this code, the "pofeaaPkg" refers to the root package of the design. The value returned by this method depends upon the value of tag "ServiceLayer." If the value of the tag is "Yes," the method returns "true," otherwise, it returns "false."

```
public static boolean needsServiceLayer() {
  boolean found = false;
  if (pofeaaPkg != null) {
    String value =
      Model.getFacade().getTaggedValueValue(pofeaaPkg,"ServiceLayer");
    if (value.equals("Yes")) found =  true;
  }
  return found;
}
```

Figure 32: A Method from GU Class which Checks "ServiceLayer" Tagged Value

The second *critic* class (`CrPatterns`), verifies the placement of patterns in the layers. In terms of the *PofEAA* rules, this class applies the remaining rules (rules 5 to 13) of the "*PofEAA* Rule Set - Part II," given in Figure 21. To fulfill its tasks, this class calls a utility method from `GU` (`GU.patternLayerMismatch(aPackage)`). In case of any error, i.e., if a pattern is located in a wrong layer, the PLA is called to report a syntactic problem and help the designer fix the problem (i.e., move the patterns to their corresponding layers). The source code of the `CrPatterns` class is attached in Appendix A.5.3.

Rules that Verify the Pattern-Pattern Relationships For checking the relationship between patterns, 15 *critics* are implemented. Some of the *critics* are at the layer level, and some are at the pattern level. Note that, there is not a one-to-one correspondence between the syntactic rules in the "*PofEAA* Rule Set - Part III," and the written *critics*. That means, some of the rules can be combined together and be checked via a single *critic*.

As an example of a syntactic *critic*, consider the following two rules from the "*PofEAA* Rule Set - Part III" (see Figure 22):

$$\textit{Domain Model} \overset{C21}{\rightarrow} \textit{Active Record} \quad \{\text{C21 : Domain Structure is Simple}\}$$

$$\textit{Domain Model} \overset{C23}{\rightarrow} \textit{Data Mapper} \quad \{\text{C23 : Domain Structure is Complex}\}$$

A *critic* named `CrDomainModelSyn` is dedicated to implement these syntactic rules. The *critic* verifies the *conditional uses* relationship between the Domain Model and the selected pattern for the Data Source Layer of the design. To do this, the *critic* verifies the consistency of the dependency between the Domain Model either to the Active Record or to the Data Mapper, subject to the complexity of the model. The *critic* applies the following criteria.

1. A Domain Model pattern is already detected by PSV, i.e., it is in PIT.
2. The Domain Model pattern is located in the Domain Layer.
3. (a) The Domain Model pattern *uses* an Active Record pattern.
   (b) The Active Record is already detected by PSV.
   (c) The Active Record pattern is located in the Data Source Layer.
   (d) The model is Simple.
4. (a) The Domain Model pattern *uses* a Domain Model pattern.
   (b) The Domain Model is already detected by PSV.
   (c) The Domain Model pattern is located in the Data Source Layer.
   (d) The model is Complex.

The criteria are checked sequentially. If any of the conditions in steps 1, 2, 3.a, 3.b, 3.c, 4.a, 4.b, or 4.c is false, the *critic* ends without triggering any error. These criteria are checked to prevent multiple error reporting. In fact, by this strategy, we are applying a type of error prioritizing which is not obvious from the rules per se. For instance, if the Active Record pattern is not structurally correct (3.b), or if it is not located in the appropriate layer (3.c), then the `CrDomainModelSyn` *critic* returns without detecting any error, because those errors should be caught by the corresponding structural *critic* (`CrActiveRecord`) or the *critic* that checks the organization of patterns (`CrLayers`). If either step 3.d or step 4.d is violated, that means there is a syntactic error in the model and it must be caught by the PTV.

Figure 33 shows an excerpt from the `CrDomainModelSyn` class which shows the "predicate" method. The source code of this class is also attached in Appendix A.5.3.

In line 6, by using the `patternFound()` method of the `GU` class, we verify that the Domain Model pattern is already detected and recorded in the PIT. Lines 7 and 8 check the containing layer of the pattern and make sure that it is the Domain Layer.

```
 1 public boolean predicate2(Object dm, Designer dsgr) {
 2   if (dm == null) return NO_PROBLEM;
 3   if (! Model.getFacade().isAClass(dm)) return NO_PROBLEM;
 4   Object dmCls = dm;
 5   if (!GU.hasStr(dmCls, "DomainModel")) return NO_PROBLEM;
 6   if (!GU.patternFound("DomainModel")) return NO_PROBLEM;
 7   Object dmPkg = Model.getFacade().getNamespace(dmCls);
 8   if (!GU.hasStr(dmPkg, "domain")) return NO_PROBLEM;
 9   Object actRec = GU.findStrSupplier(dmCls, "ActiveRecord");
10   if ( actRec != null ) {
11     if ( GU.patternFound("ActiveRecord")) {
12       Object dsPkg = Model.getFacade().getNamespace(actRec);
13       if ( GU.hasStr(dsPkg, "dataSource") )
14         if ( ! GU.hasComplexity("Simple") )
15           return PROBLEM_FOUND;
16   } }
17   Object dataMap = GU.findStrSupplier(dmCls, "DataMapper");
18   if ( dataMap != null ) {
19     if ( GU.patternFound("DataMapper")) {
20       Object dsPkg = Model.getFacade().getNamespace(dataMap);
21       if (GU.hasStr(dsPkg, "dataSource"))
22         if ( ! GU.hasComplexity("Complex") )
23           return PROBLEM_FOUND;
24   } }
25   return NO_PROBLEM;
26 }
```

Figure 33: Predicate Method of the CrDomainModelSyn Critic

To check the dependency between two classes (two patterns), there exist two utility methods in the GU class: `findStrSupplier(Object cls, String str)` and `findStrClient (Object cls, String str)`. These methods check whether there exist a supplier (or client) with stereotype "str" for a given class "cls." Using the former method, in lines 9 and 17, the dependency from the Domain Model pattern to either the Active Record or the Data Mapper is checked.

To access the tagged value "Complexity," the CrDomainModelSyn class uses the utility method `hasComplexity()` from the GU class (see lines 14 and 22). The `hasComplexity()` method works similar to the `needsServiceLayer()` shown in Figure 32. The method checks the value of tag "Complexity," and returns "true" if the value of the tag is equal to the value specified by the parameter "complexity." If the complexity of the model is not the same as what is anticipated, the *critic* triggers a syntactic error in line 15 or 23.

To summarize, PTV is implemented by two general *critics* which apply the "*PofEAA* Rule Set - Part II," plus 15 *critics* which apply the "*PofEAA* Rule Set - Part III."

4.4.6 PMV in ArgoPLV

In ArgoPLV, the PMV is built by hard coding the semantic rules of the *PofEAA* PL into the *critics*. In Section 4.2.3 we called these rules "*PofEAA* Rule Set - Part IV: Semantic Rules," represented in Figure 23. Dealing with semantic issues in the *critics* is almost similar to the syntactic ones, because most semantic *critics* need to investigate the tagged values. PMV is implemented in ArgoPLV by 10 *critics*.

As an example of a semantic *critic*, consider the following rule from Figure 23:

$$\textit{Service Layer} \approx C41 \quad \{\text{C41 : Designer wants Service Layer}\}$$

This rule implies that there is a Service Layer pattern in the model, if and only if the designer has shown his/her intention by setting the tagged value {ServiceLayer=Yes}. Therefore, the *critic* `CrServiceLayerSem` which implements this rule, must check both "if" and "only if" parts of the rule.

Remember that, in one of the syntactic *critics* (see line 18 of Figure 31), we also verify that the existence of the Service Layer is reliant on the value of the tag ServiceLayer. However, that check was only about the layer "Service Layer," and it was equivalent to the "only if" part of the above rule. The `CrServiceLayerSem` critic checks that if there exists a correct application of the Service Layer pattern inside a Service Layer package, then the value of tag ServiceLayer is "Yes," and vice versa. The same issue happens for all the supplementary layers.

One of the semantic *critics* which is more complicated than simply checking the tagged values, is `CrTableDataGatewaySem`, which implements the following rule from Figure 23:

$$\textit{Table Data Gateway} \approx \{\textit{insert() parameter list} \subseteq \textit{update() parameter list}\}$$

The *critic* must check that the list of parameters of the insert operation is a subset of the list of parameters of the update operation in the Table Data Gateway pattern. The source code of this critic is attached in Appendix A.5.4.

4.4.7 PLA in ArgoPLV

The PLA module of ArgoPLV is built via several extensions to the ArgoUML. First, the user interface of ArgoUML is extended by adding a new tab named "**Detected Patterns**" to the Details Pane of ArgoUML (see Figure 34). This tab is used to report the detected patterns (and the content of PIT) to the designer. The tab is divided into two columns. The left column is for displaying the name (Sign) of the pattern. The right column is for displaying the elements of the patterns. In the left column, two categories are defined for the detected patterns: Patterns of EAA, and Design Patterns. Obviously, detected patterns from the *PofEAA* will be placed under the first category. The second category is reserved for the GOF design patterns, in case there are *critics* for detecting them. Clicking on a pattern name, will display the pattern elements, their role, and the containing layer of the pattern in the right side of the tab.

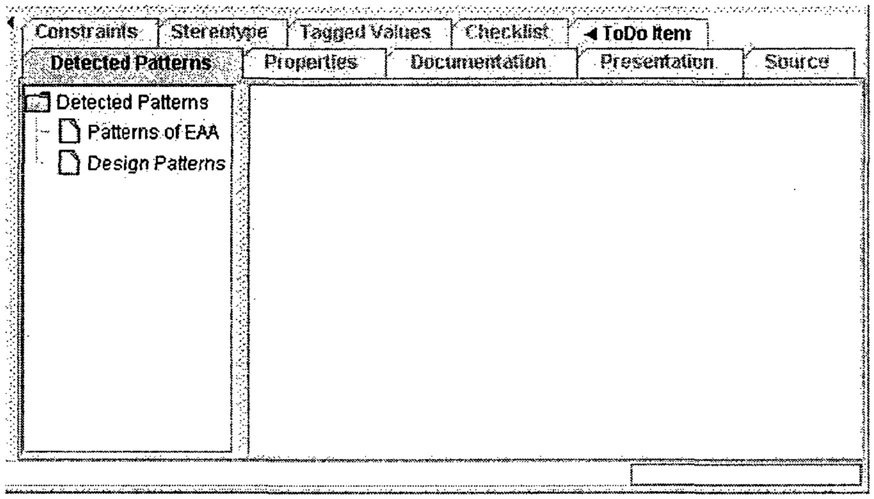
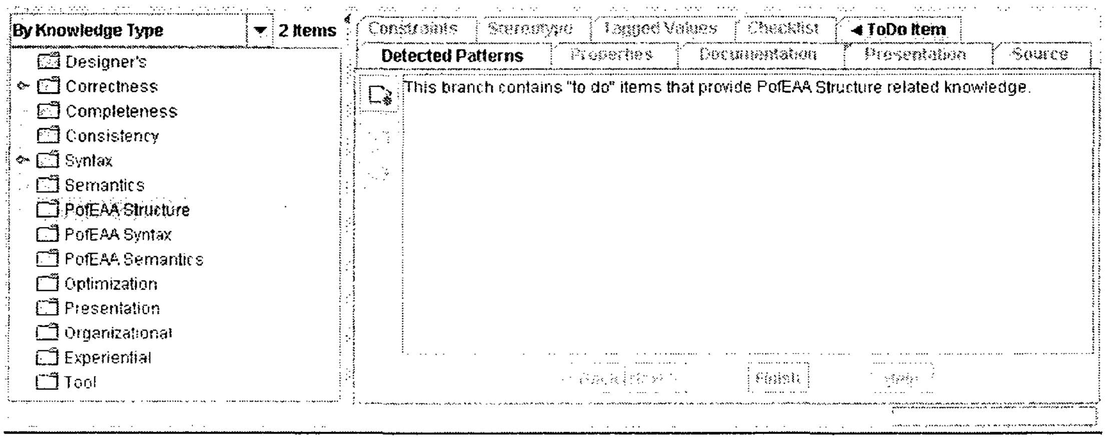


Figure 34: Detected Patterns tab is added to Details pane of ArgoUML

Second, the user interface of ArgoUML is extended by adding three new Knowledge Types in the ToDo List. The new types are ***PofEAA* Structure**, ***PofEAA* Syntax**, and ***PofEAA* Semantics**, as indicated in Figure 35. These types are created to report the corresponding three groups of errors. The errors in each group will be inserted as ToDo Items under the related Knowledge Type.



This branch contains "to do" items that provide PofEAA Structure related knowledge.

Figure 35: Three *PofEAA* Knowledge Types are added to ArgoUML's ToDo List

```
critics.CrFrontController-head =
  PofEAA: Structural Problem in using Front Controller Pattern
critics.CrFrontController-desc =
  Class "<ocl>self</ocl>" seems to be a Front Controller Pattern. Based on
  Fowler's definition, a Front Controller is a handler class for web requests.
  Therefore, it should have two operations named doGet and doPost.
  There should be a dependency between this class and another one named
  Command with at least one operation called process. The Command class
  should be abstract and have at least one child as Concrete Command.
  To address this, select "Next>" to use the wizard, or manually add the
  requested elements to the model. Note that the problems in the children
  of Command can not be fixed by this wizard.
```

Figure 36: Head and Description of the Critic Defined for the Front Controller Pattern in the "critics.properties" File

Third, for each error, the error message along with the guidelines on how to fix the problem are defined in a uniform manner, by extending the "critics.properties" file. This information is shown to the designer via ToDo Items. The guidelines show useful information that the designer can use in order to solve the problem. Figure 36 shows an excerpt from the "critics.properties" file that introduces the Head and the Description fields regarding the error message that will report a problem in applying the Front Controller Pattern.

Fourth, a Design Rationale named `PofEAA_rationale.txt` is created that keeps track of each session of the ArgoPLV by recording the actions performed by the *wizards*. There are four elements in each record of this text file: the date and the time of the decision, the name of the wizard class, the issue, and the rationale for solving the issue. The Design Rationale is very useful for people who want to work on the system in future. Table 16 shows an excerpt from the Design Rationale file. More records are shown in Appendix A.7.

Table 16: A Record from the Design Rationale File

Date Time	**Wizard Class**	**Issue**	**Rationale**
2009-01-13 15:16:24	WizTable Data-Gateway	PofEAA: Structural Problem in using Table Data Gateway Pattern	Table Data Gateway pattern needs CRUD operations. Also the return type of the Find operation should be a Record Set. This wizard has added any of those missing items to the model.

Fifth, to fulfill the most important responsibility of PLA, the *wizard* classes are written to fix the problems automatically. Automatic repair is done in a step-by-step manner which needs designer's confirmation at each step. The automatic repairs are available mainly for the structural errors. For the syntactic or semantic errors which are caused by an inappropriate value of a tagged value, changing the values of the tagged values can be done automatically, which resolves the problem. For the cases that a *wizard* is available for repairing the error, the designer can ask the *wizard* to be executed by pressing the "Next" button, after the ToDo item is displayed (see Figure 35).

In total, 50 *wizards* are written as part of the PLA module for ArgoPLV. 23 *wizards* are corresponding to 23 structural critics; Two wizards are related to `CrLayers` and `CrPatterns`; 15 wizards are defined for pattern relationship critics: 10 wizards are related to the semantic critics.

As an example, the wizard class `WizFrontController`, which fixes the structural problems of the Front Controller pattern, is shown in Appendix A.5.5. In the heart of this *wizard*, there is a method named **`fixFCProblems`** which is shown in Figure 37. This method gets the Handler class of the Front Controller pattern, the package including the pattern, and an integer "n" (line 1). The number "n" is the index of the list "misItems" which includes the list of missing items in the pattern. Depending on the missing item (the value of "misItems[n]"), one of the following actions are performed.

- Lines 3-6: If operation "doGet" or "doPost" is missing, it is added to the class.
- Lines 8-23: If the Command class is missing, a Command structure will be added to the model, including the Command class, its Concrete Command class child, and their "process" operations.
- Lines 24-25: If the Command class is not "abstract," it will set as an abstract class.
- Lines 26-29: If the "process" operation of the Command class is missing, it is added to the class.
- Lines 30-39: If the Command class has no Concrete Command child, a Concrete Command class along with the "process" operation will be added as a child to the Command class.
- Lines 40-43: If the "process" operation of the Concrete Command class is missing, it is added to the class.
- Lines 44-45: If the Concrete Command class is not specified with the corresponding stereotype, the stereotype is added to the class.

Creating the whole structure of the Command pattern (Lines 8-23) is an example of the Pattern Instantiation power of the PLA. The whole Front Controller pattern can also be instantiated this way, i.e., having a single class which has the Sign of the pattern

```
private void fixFCProblems(Object handCls, Object curPack, int n) {
  // We build doget and dopost ops in the Handler class
  if (misItems[n].equals("doGet")||misItems[n].equals("doPost")) {
    if ( ! GU.classHasSteOp(handCls, misItems[n]))
      GU.buildOpWithSte(handCls, misItems[n]+"Op", misItems[n]);
  }
  // We build a Command hierarchy and process operations
  else if ( misItems[n].equals("command") ) {
    if ( ! missingCommandCreated ) {
      Object newComClass =
        Model.getCoreFactory().buildClass("CommandCls",curPack);
      Model.getCoreFactory().buildDependency(handCls,newComClass);
      GU.addSteToObject(newComClass, "Command");
      // change Command class to Abstract
      GU.makeElementAbstract(newComClass);
      Object conComClass =
        Model.getCoreFactory().buildClass("ConcreteCommandCls",curPack);
      GU.addSteToObject(conComClass, "concretecommand");
      Model.getCoreFactory().buildGeneralization(conComClass,newComClass);
      GU.buildOpWithSte(newComClass, "processOp","process");
      GU.buildOpWithSte(conComClass, "processOp","process");
      missingCommandCreated = true;
  } }
  else if( misItems[n].equals("commandAbs") ) {
     GU.makeElementAbstract(comCls);   }
  else if ( misItems[n].equals("commandProcess") ) {
    if ( ! GU.classHasSteOp(comCls, "process")) {
        GU.buildOpWithSte(comCls, "processOp","process");
  } }
  else if ( misItems[n].equals("commandChildren") ) {
    if ( ! missingConCommandCreated ) {
      Object conComClass =
        Model.getCoreFactory().buildClass("ConcreteCommand",curPack);
      GU.addSteToObject(conComClass, "concretecommand");
      Model.getCoreFactory().buildGeneralization(conComClass,comCls);
      if ( ! GU.classHasSteOp(conComClass, "process"))
        GU.buildOpWithSte(conComClass, "processOp","process");
      missingConCommandCreated = true;
  } }
  else if ( misItems[n].equals("conCommandProcess") ) {
    if ( ! GU.classHasSteOp(conComClass, "process")) {
        GU.buildOpWithSte(conComClass, "processOp","process");
  } }
  else if ( misItems[n].equals("conCommandSte") )
    GU.addSteToObject(conComClass, "ConcreteCommand");
}
```

Figure 37: An Excerpt from the Front Controller Wizard

(«FrontController») on it, causes all the above repair steps take place and an instance of the whole pattern is created.

4.4.8 Using ArgoPLV

How does the ArgoPLV tool help a designer in applying the *PofEAA*? ArgoUML, and hence ArgoPLV, is an interactive modeling tool. By applying the appropriate stereotypes, the designer shows his/her intention in using a pattern (remember that only one stereotype is considered as the "sign" for identifying a pattern). Immediately after applying the sign stereotype, the corresponding *critic* is activated and verifies the structure of the pattern (PSV module). If any of the structural criteria fail, the *critic* is triggered and a ToDo Item (*critique*) will be posted in the ToDo List under ***PofEAA* Structure**. By selecting a ToDo Item, its description will be shown in the Details Pane, and upon the user's request, the *wizard* for the *critic* will be executed and the problems found in the pattern usage will be fixed (PLA module). The details of the correctly applied patterns (PIT content) is displayed in the **Detected Patterns** tab in the Details Pane.

If a syntactic problem is detected in the pattern combinations (by the PTV module), one of the syntactic *critics* is triggered and a ToDo Item (*critique*) will be posted in the ToDo List under ***PofEAA* Syntax**. If any of the semantic criteria fail, e.g., an inconsistency between the design with context information is caught, one of the semantic *critics* is triggered (by the PMV module), and a ToDo Item (*critique*) will be posted in the ToDo List under **PofEAA Semantics**. In either of the cases, the *wizards* might be available to fix the problem automatically, or the designer is guided to repair the error manually.

In the next chapter, a real application is designed using the ArgoPLV. It is shown how the tool is able to help a novice designer improve his/her design.

4.5 Discussion

4.5.1 Summary

This chapter aimed to show how to implement a Pattern Language Verifier (PLV), for an existing Pattern Language (PL), through a case study. To evaluate the idea of PLV and its applicability and usefulness in current modeling tools, we did experiments with the ArgoUML modeling tool. Using the idea of the PLV process, we defined a PLV for *PofEAA* as an integration into ArgoUML, named ArgoPLV. To make this case study simple and concrete, we selected 23 out of 51 patterns of *PofEAA*. We discussed the steps of building ArgoPLV as a PLV for *PofEAA* PL. We observed that the PLV process is able to be integrated in ArgoUML by writing Java code. However, hard coding the process into the tool is not a convenient way of tool extension and impedes the scalability of the process.

Also when a rule is not accurate, it causes ambiguity in implementation, and hence in detecting errors in a model.

To summarize, the main steps in defining a PLV for a PL are:

1. Extract the rules, or even the informal advices, of the PL that govern the structural, syntactic, and semantic aspects of the language.

2. Build the Rule Set of the PL using the formalisms proposed in Chapter 3.

3. Define a UML profile for the PL.

    (a) Build a domain model (metamodel) for the PL. Patterns are the principal concepts in this domain model.

    (b) Map that domain model into the UML metamodel.

    (c) Define the stereotypes; Define the tagged values for each stereotype.

    (d) Define the constraints (inspired by the Rule Set obtained in step 2). There are two alternatives for the constraints: First, to interpret the Rule Set into OCL constraints; Second, to accept the Rule Set as the constraints.

4. Build the PLV modules. There are two alternatives for building modules depending on the previous step.

    (a) For OCL constraints: PSV, PTV, and PMV modules are obtained by hand-coding the checking of the profile constraints. The PLA module must be implemented separately!

    (b) For accepting Rule Set as the constraints: All modules of PLV are implemented as a modeling tool, or as extension to an existing modeling tool.

4.5.2 Issues Related to Building a PLV

Issues for Step 1 An important issue is to classify the rules into appropriate groups. Some advices/rules can be considered both syntactic and semantic, e.g., advice A09 from *PofEAA* (see Appendix A.2) is: "A domain layer that uses only Transaction Script isn't complex enough to warrant a separate [Service] layer." This advice is twofold: it can be interpreted as a syntactic rule that a *uses* relationship exists from a Service Layer pattern to a Transaction Script pattern *or* it can be interpreted as a semantic rule that existence of Service Layer is inconsistent with setting {Complexity=Low}.

Issues for Step 2 Some advices are semantic rules that need linguistic checks, e.g., consider advice A21 (see Appendix A.2): "With a Domain Model we build a model of our domain which, at least on a first approximation, is organized primarily around the nouns in the domain." In order to apply this advice as a semantic rule, we must verify that the domain objects' names are the nouns in the domain.

Issues for Step 3 If there is no tree or graph in the language that shows pattern dependencies, then deriving a domain model for it is hard.

Issues for Step 4 For Step 4.a, we experienced using the Object Constraint Language Environment (OCLE) [CPC$^+$04] for implementing some of the structural rules of the *PofEAA* [ZB07]. OCLE [Uni09a] as a UML CASE tool, offers many useful features including OCL support at both UML metamodel and model level, and a graphical interface for creating UML diagrams.

In OCLE, Users are able to compile and run the constraints against the models. A Compile-time error reflects problems concerning OCL syntax. A Runtime error means that some of the invariants in constraints are violated. In this case, a message is displayed to the user and it is the user's responsibility to fix the error.

For Step 4.b, implementation could be a laborious task. For instance, to check the dependency between a Handler class (as a client) and a Command class (as a supplier) in the Front Controller pattern (see Appendix A.3.1), we need to check all the dependencies that may exist from the Handler to other classes, then for each of the dependencies, we should find the collection of Suppliers, then for each supplier class, we should check the collection of stereotypes, then if at least one of the stereotypes satisfies the condition (e.g., is «Command»), then we make sure that we have found the Command class!

As another environment for verifying the constraints, we have experienced working with Epsilon Wizard Language (EWL) (EWL is part of Epsilon Object Language (EOL)) for implementing the PSV [ZB07].

4.5.3 Other Issues

Considering the PLV process, since the three verifying modules perform model independent tasks and need to be verified against the metamodel, their tasks is done using OCL in metamodel level constraint files. However, due to the lack of capability for model modifications by OCL, there is no support for the tasks of PLA. It is up to the user to check every invariant and, for every failed invariant, the user should fix the cause of the problem. The problem here is how to synchronize the PLA with other modules. Finally, the "*PofEAA* UML Profile" will provide the novice designers with great assistance on how to break the system into layer, how to select appropriate patterns for each layer, how to use the patterns, and how to maintain a good structure for their design.

Chapter 5

ArgoPLV in Action

This chapter shows how the ArgoPLV can be utilized as a modeling tool in a real situation. For this purpose, we need to consider a sample application which is going to be designed based upon the Patterns of Enterprise Application Architecture (*PofEAA*).

Section 5.1 introduces the application: an *Online Student Registration System.* In Section 5.2, we demonstrate how ArgoPLV is used as a design critiquing tool in a step-by-step design of the application using the patterns of *PofEAA*. Section 5.3 shows how the ArgoPLV can be used to verify the application of the *PofEAA* Pattern Language (PL) in a given design model of the application. Section 5.4 discusses the validation issues.

5.1 The Application

We consider a simple *Online Student Registration System* as our sample application. The system consists of students, professors, courses, and departments. Each student studies in one department. Only the research students (thesis-based) should have one of the professors as their supervisors. A student can take a course if he/she has already passed its prerequisites. Each course, is offered by one department, is taught by one professor, and may have many prerequisites. One professor works for one department. A professors' job is to teach courses and supervise students. For students and professors, the personal information and the address is recorded in the system. Figure 38 shows the domain model of this system by a UML class diagram.

The application is a web-based online registration system that allows persons (both students and professors) to enter or edit their personal information. Professors can select courses for teaching. Students can register for courses by filling in an online registration form. Only a research student can request a professor to be his/her supervisor, which needs

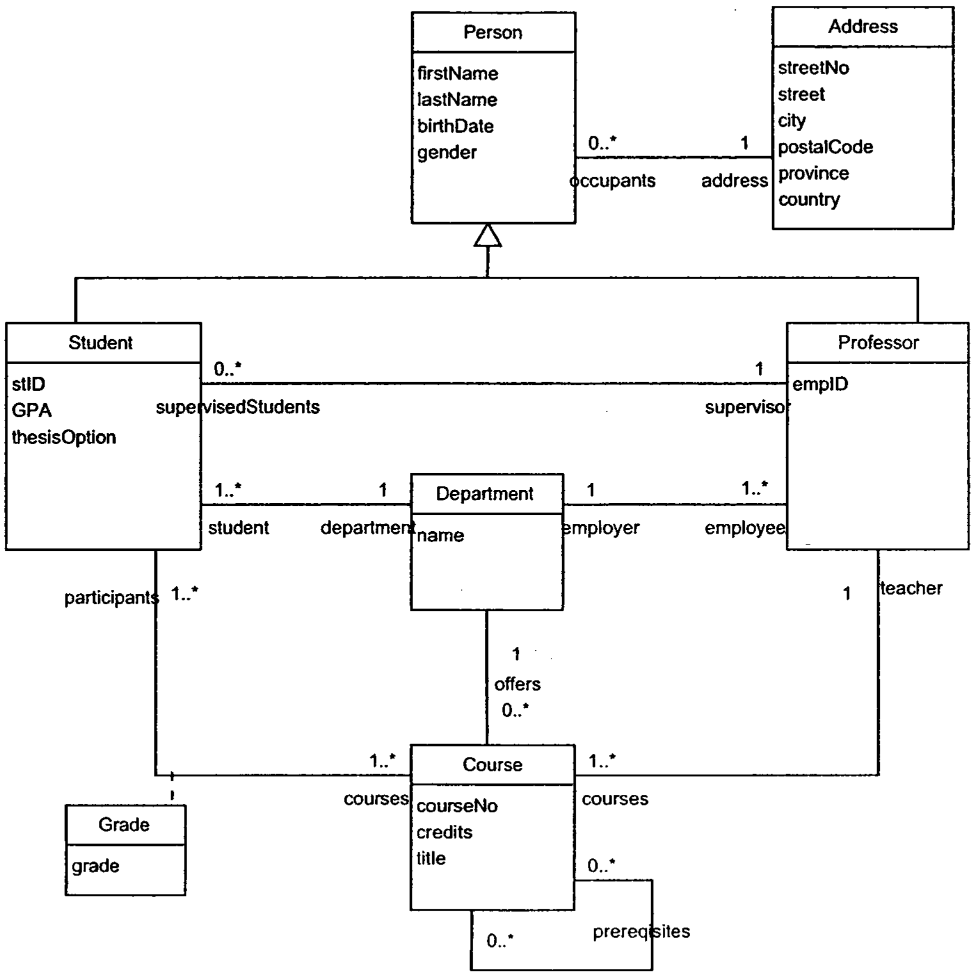


Figure 38: Domain Model of the Online Student Registration System

the professor's confirmation.

The system must provide a variety of appropriate reports for each user. Students can get the following reports: list of offered courses and their prerequisites, list of courses taken, up-to-date transcript and GPA (Grade Point Average). For professors, list of offered courses, list of registered students, and list of supervised students are important reports.

The system must be secured by providing each user a user-id and password to enter the system. The users are able to change their passwords at any time. There are different levels of users who can access the system, e.g., students and professors. Students are able to modify their personal information only. They can browse the professor and course information, when they decide to register for the current term. Professors have read access to all information regarding their students, but they can enter the grades.

The reliability, availability, and consistency of persistent data is a very important requirement of the system. The system transactions must be atomic and consistent. Concurrency control should be performed in order to prevent loss or inconsistency of information. An attempt should be made to make the system available all the time. Such a system needs a DBMS for file management. Due to the fact that, in our *PofEAA* selected patterns, we excluded all the Object-Relational patterns, we ignore about the DB issues in the remaining parts of this chapter. Considering the above requirements, we select the following features for this case study, categorized by the user of the feature.

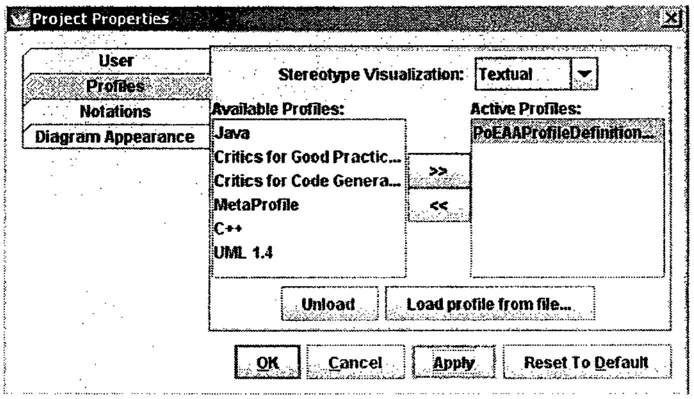
1. Features that are particularly defined for the students:
   (a) Browse Courses
   (b) Register Course
   (c) Browse Professors
   (d) View Professor
   (e) Request Supervision
   (f) Calculate GPA
2. Features that are particularly defined for the professors:
   (a) Select Course
   (b) Browse Students
   (c) Browse Supervised Students
   (d) Browse Supervision Requests
   (e) View Supervision Request
   (f) Accept Supervision Request
   (g) Enter Grades
3. Features that are common for both students and professors:
   (a) Login and Logout the System
   (b) Edit Personal Information
   (c) Browse Courses
   (d) View Course
   (e) Check Course Prerequisites
   (f) Send List of Courses to Other University

5.2 Using ArgoPLV in Stepwise Design of the Application

In this section, we show how the ArgoPLV tool helps a designer build a model for the *Online Student Registration System* based upon the patterns of the *PofEAA* PL. We walk through a scenario and discuss the step-by-step design of the system. For each step, a screen shot (maybe partial) of the ArgoPLV is shown. Between the steps, there are paragraphs that discuss the errors caught by the ArgoPLV, the guidelines given to the designer, and the repairs done to the model.

Step 1: Create Model Designer creates a project named `University`. Inside the project, he/she creates a model named `UniversityModel`.

Step 2: Apply Profile Designer applies the *PofEAA* UML profile on the model (see Figure 39).


Stereotype Visualization: Textual

Available Profiles:
Java
Critics for Good Practic...
Critics for Code Genera...
MetaProfile
C++
UML 1.4

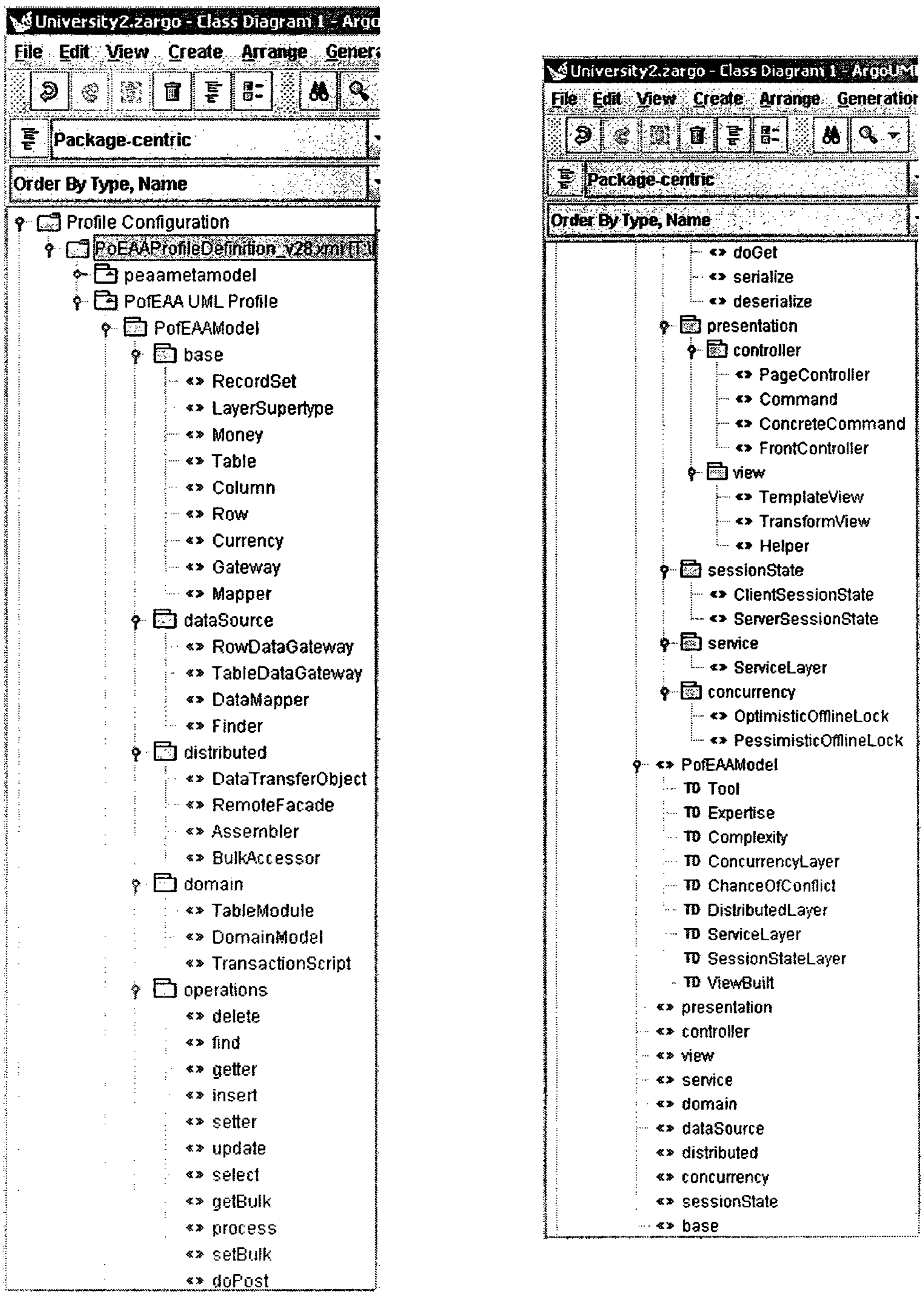
Active Profiles:
PoEAAProfileDefinition...

Figure 39: Applying *PofEAA* UML Profile on the Model

Step 3: Explore Stereotypes Designer explores the applicable stereotypes and tagged values of the *PofEAA* profile in the Explorer Pane (see Figure 40).

Step 4: Specify *PofEAA* Model and Context Information The designer indicates his/her intention of designing a system based upon the patterns of the *PofEAA* by setting the stereotype «PofEAAModel» on a root package. Then, the designer specifies the context information, by setting the tagged values for the stereotype «PofEAAModel» (see Figure 41).

Syntactic Problem Detection Regarding the Organization of Patterns Pattern Language Syntactic Verifier (PTV) detects syntactic problems in the model, due to the fact


University2.zargo - Class Diagram 1 - Argo
File Edit View Create Arrange Gener
Package-centric
Order By Type, Name

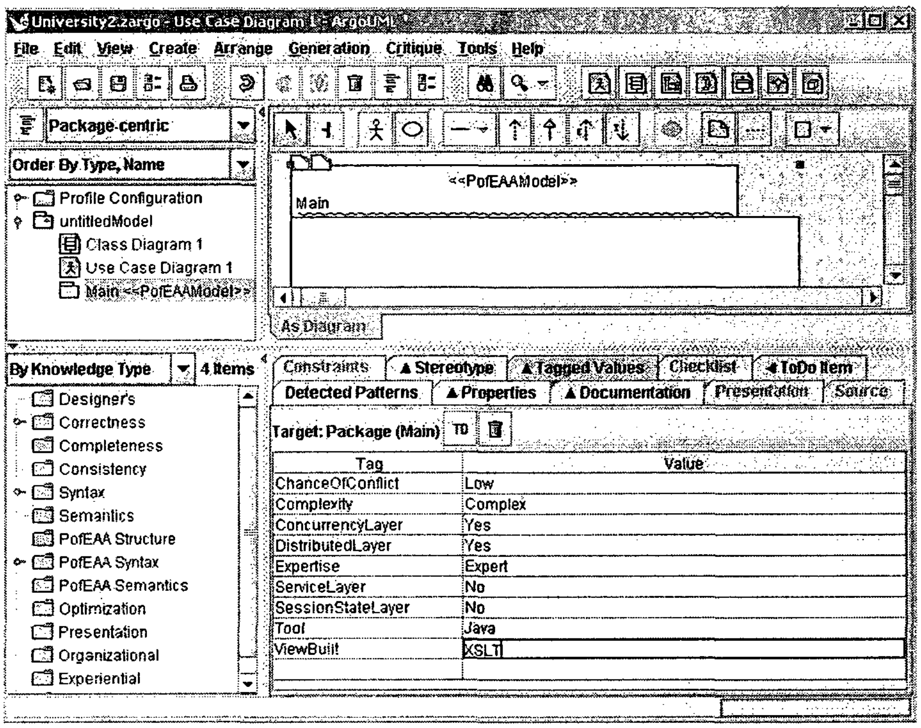
```
Profile Configuration
  PoEAAProfileDefinition_v28.xmi
    peaametamodel
    PofEAA UML Profile
      PofEAAModel
        base
          «» RecordSet
          «» LayerSupertype
          «» Money
          «» Table
          «» Column
          «» Row
          «» Currency
          «» Gateway
          «» Mapper
        dataSource
          «» RowDataGateway
          «» TableDataGateway
          «» DataMapper
          «» Finder
        distributed
          «» DataTransferObject
          «» RemoteFacade
          «» Assembler
          «» BulkAccessor
        domain
          «» TableModule
          «» DomainModel
          «» TransactionScript
        operations
          «» delete
          «» find
          «» getter
          «» insert
          «» setter
          «» update
          «» select
          «» getBulk
          «» process
          «» setBulk
          «» doPost
```

University2.zargo - Class Diagram 1 - ArgoUML
File Edit View Create Arrange Generation
Package-centric
Order By Type, Name

```
          «» doGet
          «» serialize
          «» deserialize
        presentation
          controller
            «» PageController
            «» Command
            «» ConcreteCommand
            «» FrontController
          view
            «» TemplateView
            «» TransformView
            «» Helper
        sessionState
          «» ClientSessionState
          «» ServerSessionState
        service
          «» ServiceLayer
        concurrency
          «» OptimisticOfflineLock
          «» PessimisticOfflineLock
      «» PofEAAModel
        TD Tool
        TD Expertise
        TD Complexity
        TD ConcurrencyLayer
        TD ChanceOfConflict
        TD DistributedLayer
        TD ServiceLayer
        TD SessionStateLayer
        TD ViewBuilt
      «» presentation
      «» controller
      «» view
      «» service
      «» domain
      «» dataSource
      «» distributed
      «» concurrency
      «» sessionState
      «» base
```

Figure 40: Exploring Stereotypes and Tagged Values of *PofEAA* UML Profile

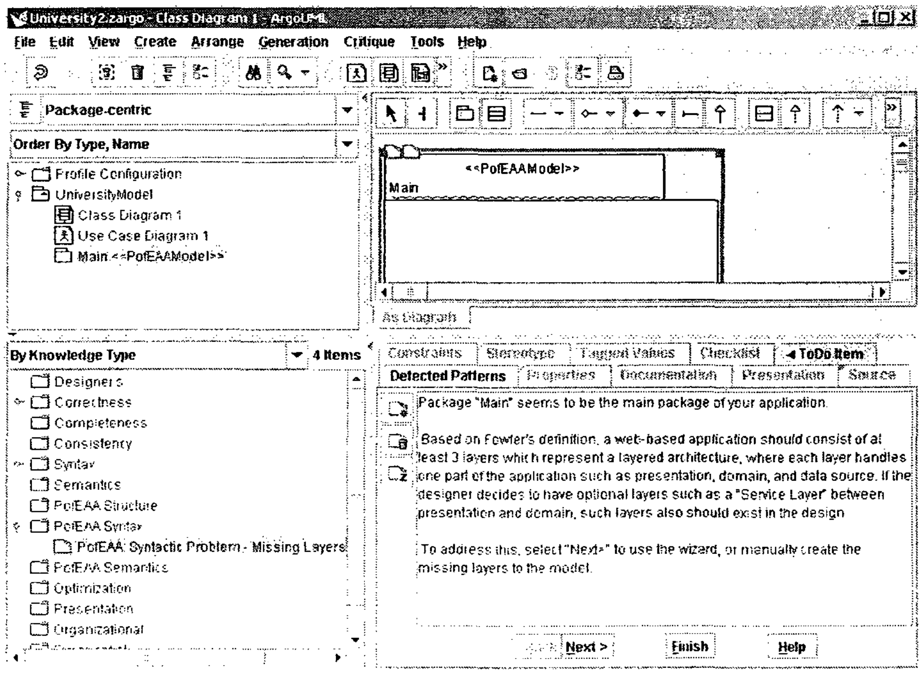
that there are missing layers (both principal and optional) in the design. Pattern Language Advisor (PLA) reports the problem to the designer, by posting a ToDo Item (critique) in the "PofEAA Syntax" category of the ToDo List (see the ToDo Pane, lower-left, in Figure 42). PLA shows the guidelines to the designer (see the Details Pane, lower-right, in Figure 42).



Target: Package (Main)

Tag	Value
ChanceOfConflict	Low
Complexity	Complex
ConcurrencyLayer	Yes
DistributedLayer	Yes
Expertise	Expert
ServiceLayer	No
SessionStateLayer	No
Tool	Java
ViewBuilt	XSLT

Figure 41: Setting the Stereotype and Tagged Values of the Main Package



PofEAA Syntactic Problem - Missing Layers

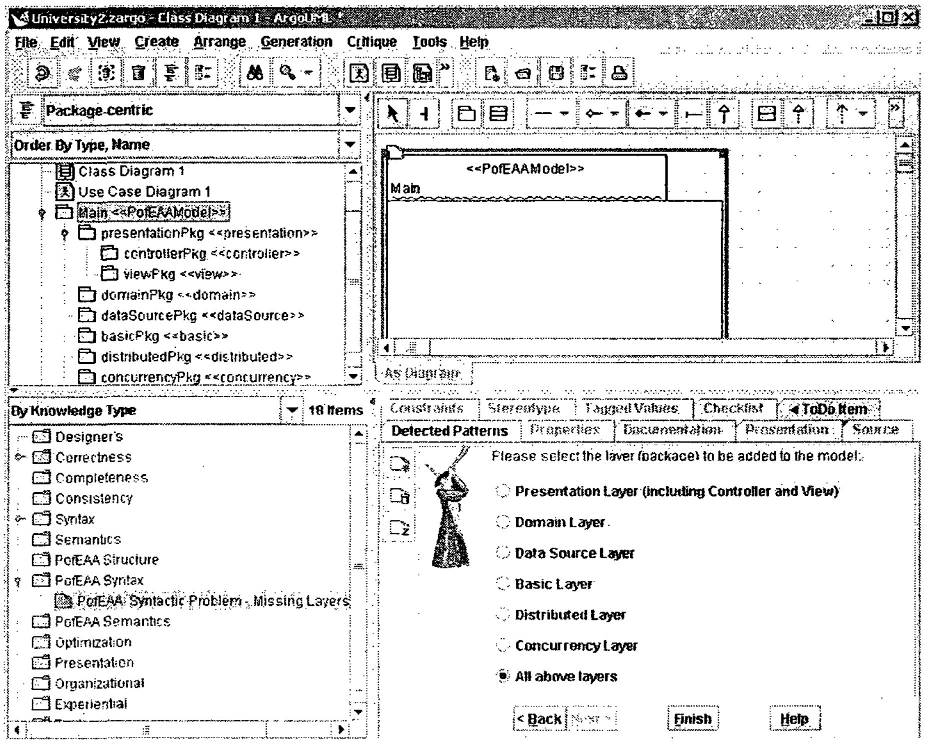
Package "Main" seems to be the main package of your application.

Based on Fowler's definition, a web-based application should consist of at least 3 layers which represent a layered architecture, where each layer handles one part of the application such as presentation, domain, and data source. If the designer decides to have optional layers such as a "Service Layer" between presentation and domain, such layers also should exist in the design

To address this, select "Next>" to use the wizard, or manually create the missing layers to the model.

Figure 42: Reporting Syntactic Problem and Showing Guidelines to the Designer

Syntactic Problem Repair The designer asks help from the PLA (by pressing the "Next" button of the Details Pane). PLA shows the options (name of missing layers) to the designer (see the Details Pane in Figure 43), and upon his/her confirmation, adds all the missing layers (three principal and three supplementary) to the model (see the Explorer Pane, top-left, in Figure 43). Note that the elements that are added by the PLA to the model, are shown only in the Explorer Pane, but they are not shown in the Editor Pane (see the Editor Pane, top-right, in Figure 43). To see these elements in the Editor Pane, the designer has to drag and drop them into the Editor Pane.


University2.zargo - Class Diagram 1 - ArgoUML *
File Edit View Create Arrange Generation Critique Tools Help
Package-centric
Order By Type, Name
By Knowledge Type
18 Items
As Diagram
Constraints Stereotype Tagged Values Checklist ToDo Item
Detected Patterns Properties Documentation Presentation Source
<< Back Next > Finish Help

Class Diagram 1
Use Case Diagram 1
Main <<PofEAAModel>>
presentationPkg <<presentation>>
controllerPkg <<controller>>
viewPkg <<view>>
domainPkg <<domain>>
dataSourcePkg <<dataSource>>
basicPkg <<basic>>
distributedPkg <<distributed>>
concurrencyPkg <<concurrency>>

<<PofEAAModel>>
Main

Designer's
Correctness
Completeness
Consistency
Syntax
Semantics
PofEAA Structure
PofEAA Syntax
PofEAA Syntactic Problem - Missing Layers
PofEAA Semantics
Optimization
Presentation
Organizational
Experiential

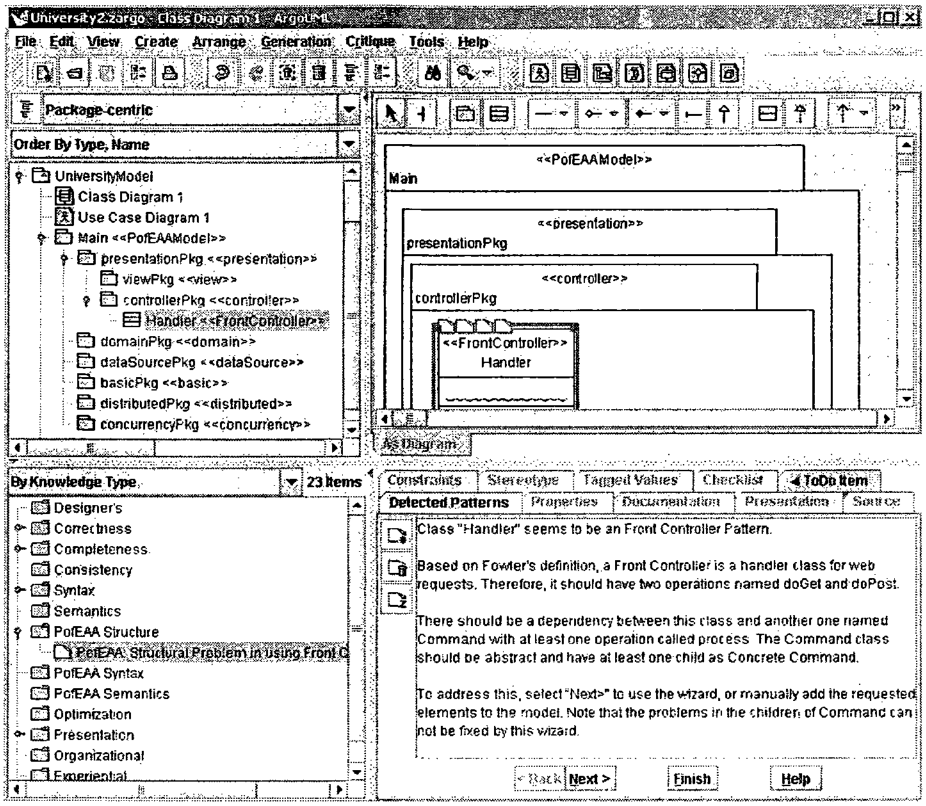
Please select the layer (package) to be added to the model:

- ( ) Presentation Layer (including Controller and View)
- ( ) Domain Layer
- ( ) Data Source Layer
- ( ) Basic Layer
- ( ) Distributed Layer
- ( ) Concurrency Layer
- [✓ All above layers]

Figure 43: Automatic Fix of Syntactic Problem by Adding all Missing Layers

Step 5: Design the Presentation Layer (Controller Part) The designer intends to apply the Front Controller pattern as the controller part of the presentation layer. However, he/she does not know the structure of this pattern exactly. Hence, he/she creates a class named "Handler" in the Controller Layer and applies the «FrontController» stereotype on it, and leaves the pattern instantiation to the ArgoPLV (see the Editing Pane in Figure 44).

Structural Problem Detection Pattern Structural Verifier (PSV) detects the structural problems (missing elements) in the application of the Front Controller pattern. PLA reports the problem to the designer, by posting a critique in the "PofEAA Structure" category of



Class "Handler" seems to be an Front Controller Pattern.

Based on Fowler's definition, a Front Controller is a handler class for web requests. Therefore, it should have two operations named doGet and doPost.

There should be a dependency between this class and another one named Command with at least one operation called process. The Command class should be abstract and have at least one child as Concrete Command.

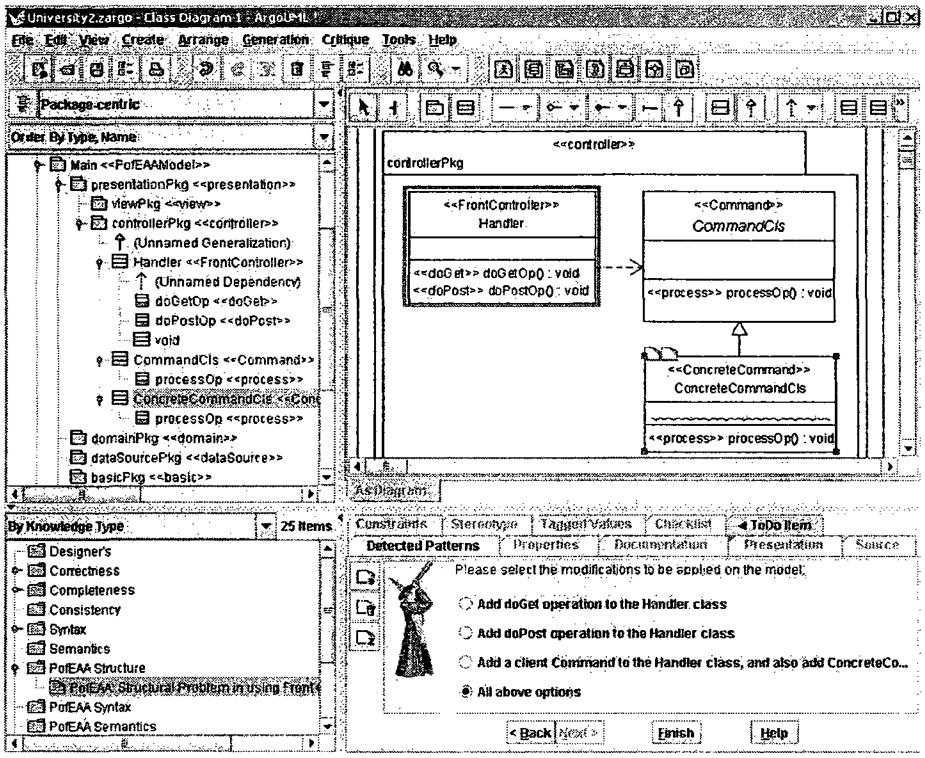
To address this, select "Next>" to use the wizard, or manually add the requested elements to the model. Note that the problems in the children of Command can not be fixed by this wizard.

Figure 44: Applying the Front Controller Pattern

the ToDo List (see the ToDo Pane in Figure 44). PLA shows the guidelines on how to fix the problem to the designer (see the Details Pane in Figure 44).

Structural Problem Repair After designer asks help from the PLA, it shows the repair options to the designer (see the Details Pane in Figure 45). By selecting the "All above options," the designer gives permission to the PLA to add all the missing parts of the pattern automatically to the model. PLA adds the missing elements and their relationships to the Control Layer of the model. To make the added classes visible, the designer drags and drops them into the Editor Pane (see both Explorer Pane and Editing Pane in Figure 45).

Step 6: Design the Presentation Layer (View Part) The designer selects the Template View pattern to format the web pages of the application. Therefore, he/she applies the «TemplateView» stereotype on a View class in the View Layer of the model. Similarly to what happened for the Controller part, the PSV detects the structural problems in the application of the Template View pattern. PLA reports the problem to the designer, and upon designer's request, PLA adds a Helper class as a supplier to the View class to fix the



Please select the modifications to be applied on the model:

- Add doGet operation to the Handler class
- Add doPost operation to the Handler class
- Add a client Command to the Handler class, and also add ConcreteCo...
- [✓ All above options]

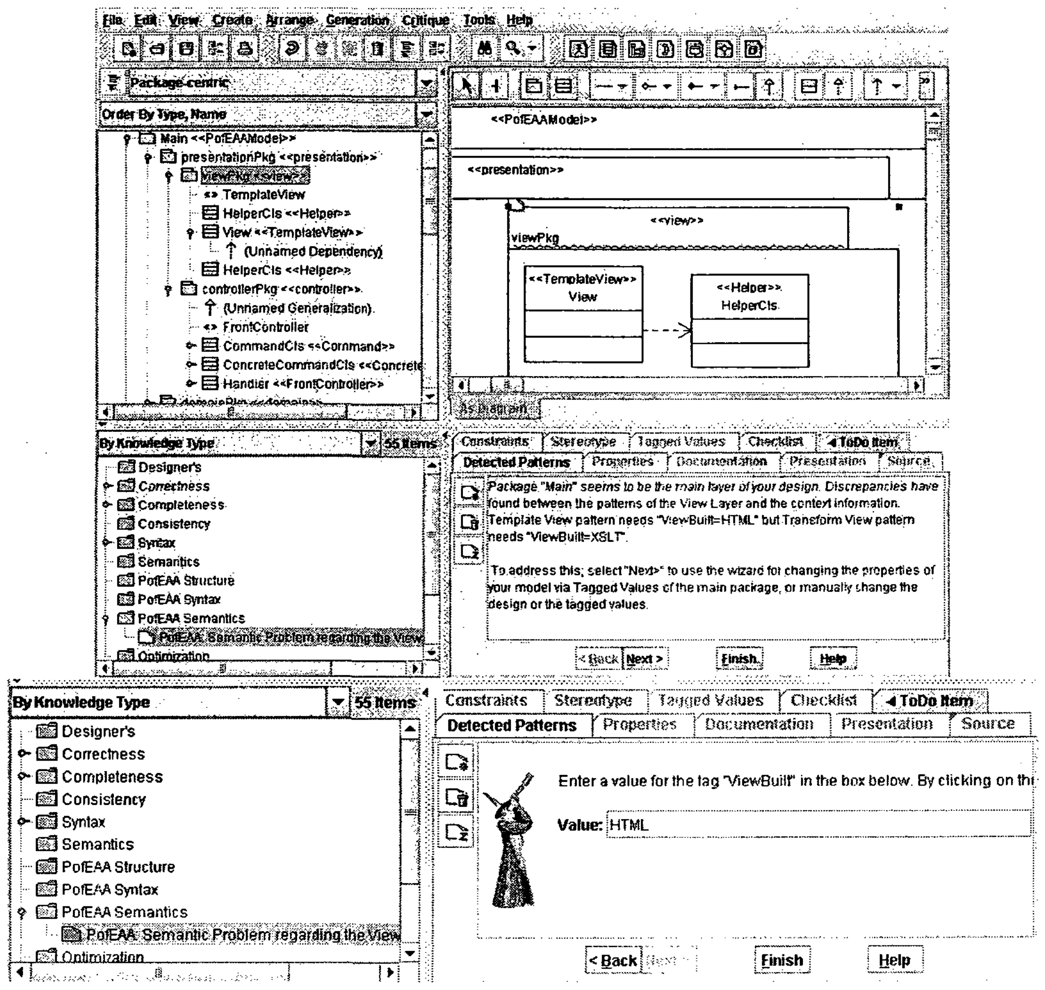
Figure 45: Automatic Fix of Structural Problems in Applying Front Controller

structural error automatically (see the Editor Pane inFigure 46).

Semantic Problem Detection Pattern Language Semantic Verifier (PMV) detects a semantic problem in the design, due to the inconsistency between the context information (the tagged value "{ViewBuilt = XSLT}") and the application of the Template View pattern. PLA reports the problem to the designer, by posting a critique in the "PofEAA Semantics" category of the ToDo List (see the upper ToDo Pane in Figure 46). PLA shows the guidelines to the designer (see the upper Details Pane in Figure 46).

Semantic Problem Repair After the designer asks help from the PLA, it shows a text box with a default value "HTML" for the tag "ViewBuilt" to the designer (see the lower Details Pane in Figure 46). The designer accepts the value by pressing the "Finish" button which results in disappearing of the semantic error.

Step 7: Review the Detected Patterns The designer wants to know which layers exist and which patterns are applied in the current model. The layers that exist in the current design model, and the patterns that are applied correctly, are presented under the "Patterns of EAA" category in the **Detected Patterns** tab in the Details Pane.



Package "Main" seems to be the main layer of your design. Discrepancies have found between the patterns of the View Layer and the context information. Template View pattern needs "ViewBuilt=HTML" but Transform View pattern needs "ViewBuilt=XSLT".

To address this; select "Next>" to use the wizard for changing the properties of your model via Tagged Values of the main package, or manually change the design or the tagged values.

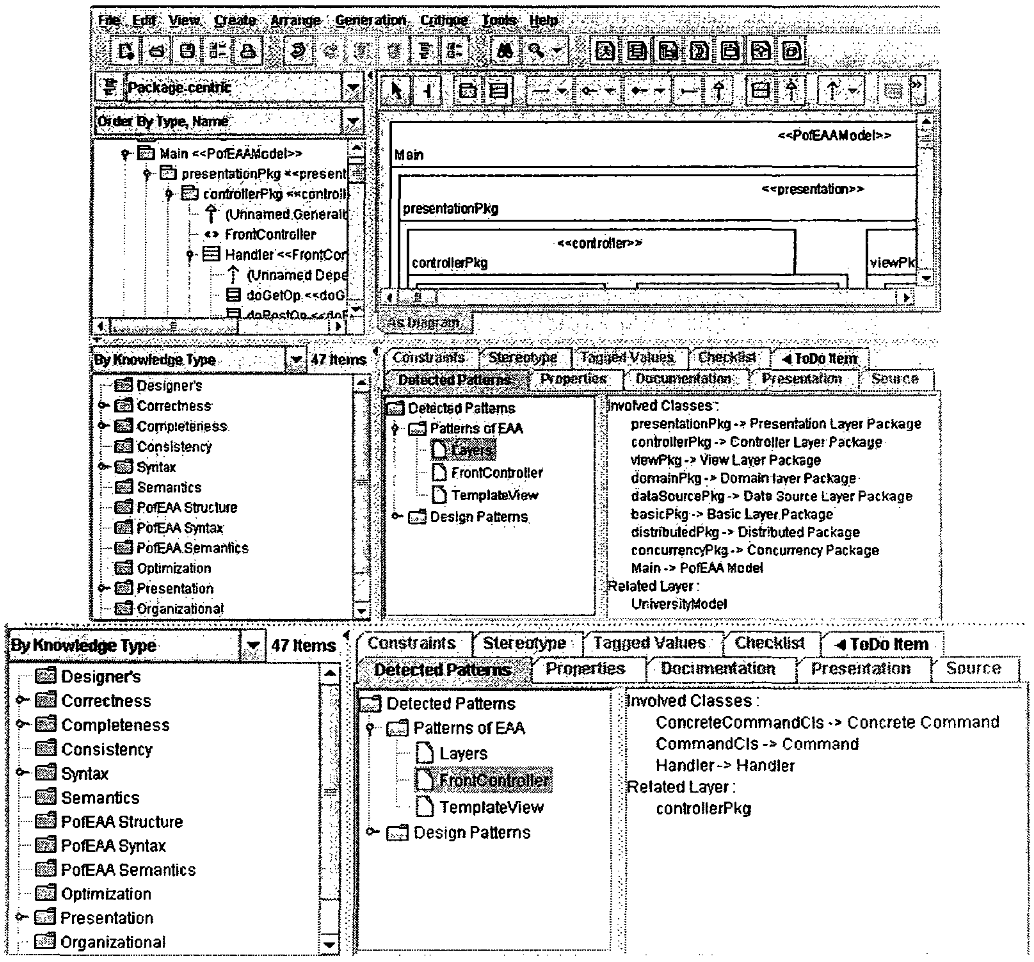
Enter a value for the tag "ViewBuilt" in the box below. By clicking on th

Value: HTML

Figure 46: Automatic Fix of Semantic Problem in Applying Template View Pattern

The detected layers are shown in the upper Details Pane in Figure 47. The detected patterns are shown in the lower Details Pane in Figure 47. Selecting a pattern from the list, causes the involved classes and the containing layer of the pattern to be shown in the right window. In Figure 47, the Front Controller pattern is selected and its information is shown.

Step 8: Design the Domain Layer The designer selects the Domain Model pattern to structure the domain logic of the application. Therefore. he/she draws a class diagram corresponding to the domain model of the system in the Domain Layer of the model, and then applies the «DomainModel» stereotype on all the classes. e.g.. the class Student. Based on the criteria of this pattern, each class must have at least one attribute and one operation (see the Editor Pane in Figure 48).



Main <<PofEAAModel>>
presentationPkg <<present
controllerPkg <<controll
(Unnamed Generali
FrontController
Handler <<FrontCon
(Unnamed Depe
doGetOp <<doG

<<PofEAAModel>>
Main
<<presentation>>
presentationPkg
<<controller>>
controllerPkg
viewPk

Designer's
Correctness
Completeness
Consistency
Syntax
Semantics
PofEAA Structure
PofEAA Syntax
PofEAA Semantics
Optimization
Presentation
Organizational

Detected Patterns
Patterns of EAA
Layers
FrontController
TemplateView
Design Patterns

Involved Classes:
presentationPkg -> Presentation Layer Package
controllerPkg -> Controller Layer Package
viewPkg -> View Layer Package
domainPkg -> Domain layer Package
dataSourcePkg -> Data Source Layer Package
basicPkg -> Basic Layer Package
distributedPkg -> Distributed Package
concurrencyPkg -> Concurrency Package
Main -> PofEAA Model
Related Layer:
UniversityModel

Designer's
Correctness
Completeness
Consistency
Syntax
Semantics
PofEAA Structure
PofEAA Syntax
PofEAA Semantics
Optimization
Presentation
Organizational

Detected Patterns
Patterns of EAA
Layers
FrontController
TemplateView
Design Patterns

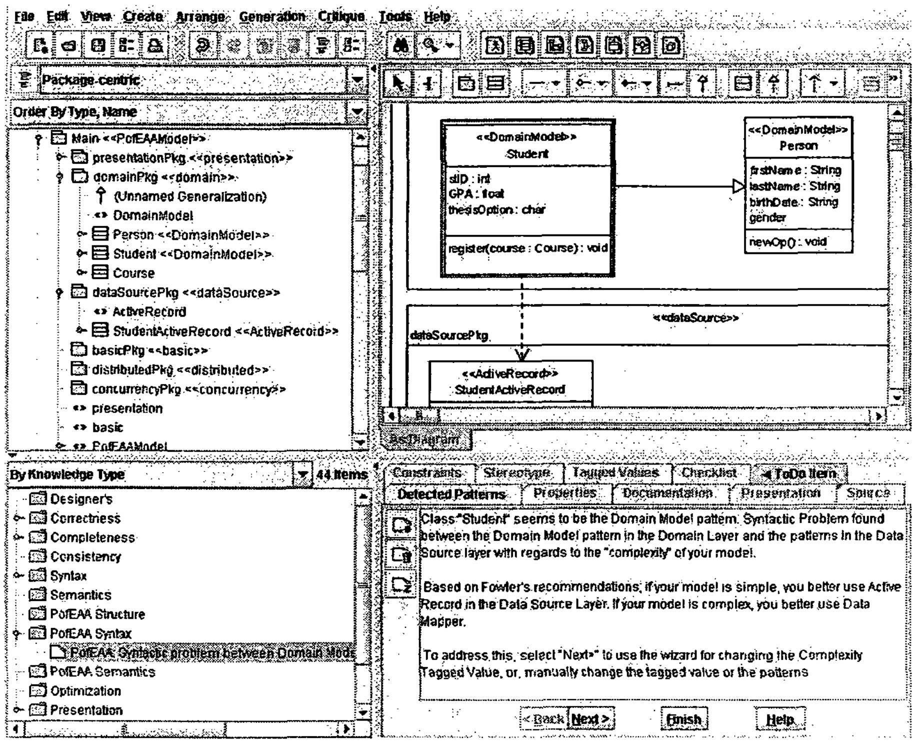
Involved Classes:
ConcreteCommandCls -> Concrete Command
CommandCls -> Command
Handler -> Handler
Related Layer:
controllerPkg

Figure 47: Reporting the Detected Layers and Patterns of the Design Model

Step 9: Design the Data Source Layer The designer decides to apply the Active Record pattern for accessing the Student record in the Data Source Layer. Therefore, he/she applies the stereotype «ActiveRecord» on a class named "StudentActiveRecord" in the Data Source Layer which is a supplier class for the "Student" class of the Domain Layer (see the Editor Pane in Figure 48).

Syntactic Problem Detection Regarding the Relationship Between Patterns A syntactic error is detected by the PTV due to the problematic relationship between the Domain Model pattern of the Domain Layer with the Active Record pattern of the Data Source Layer (see the ToDo Pane in Figure 48). The problem is also affected by the Context Information which is set by the designer in step 4. Remember that complexity of the domain was set to "Complex." The PLA shows the guidelines that suggests using Data Mapper instead of Active Record or changing the choice for complexity (see the Details Pane in

Figure 48).



Class "Student" seems to be the Domain Model pattern. Syntactic Problem found between the Domain Model pattern in the Domain Layer and the patterns in the Data Source layer with regards to the "complexity" of your model.

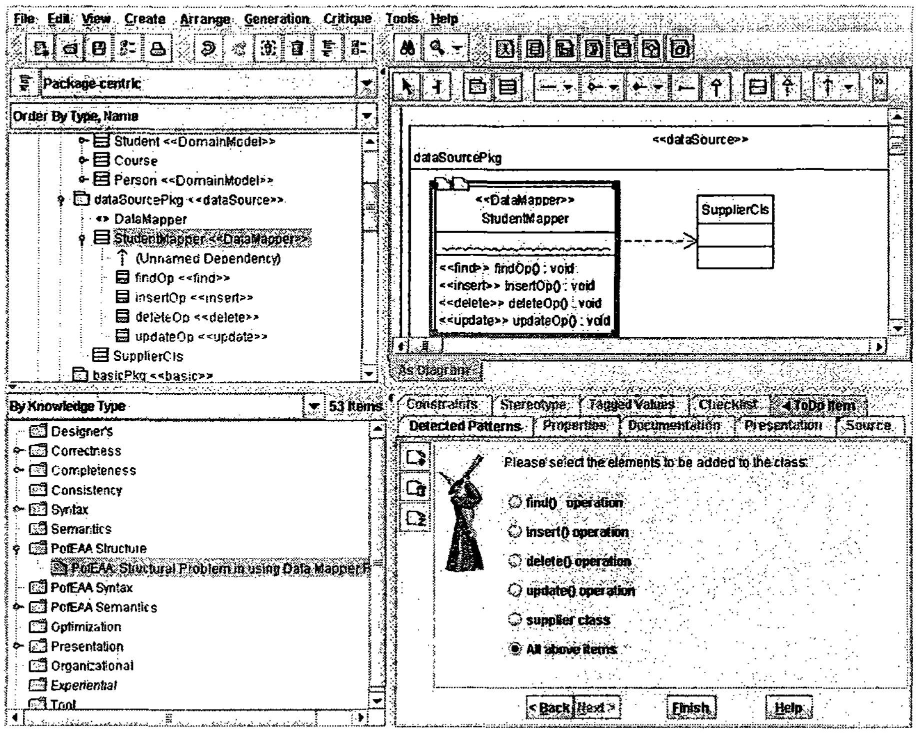
Based on Fowler's recommendations, if your model is simple, you better use Active Record in the Data Source Layer. If your model is complex, you better use Data Mapper.

To address this, select "Next>" to use the wizard for changing the Complexity Tagged Value, or, manually change the tagged value or the patterns

Figure 48: Reporting the Syntactic Problem Regarding the Relationship between the Domain Model pattern and the Active Record pattern

Step 10: Syntactic Problem Repair by Changing the Design of Data Source Layer The designer has two options to fix the detected syntactic problem: either to change the value of the tag "Complexity" from "Complex" to "Simple," or to change the pattern used in the Data Source Layer. He/she decides to change the pattern for accessing the Student record in the Data Source Layer from Active Record to Data Mapper. Hence, he/she removes the "StudentActiveRecord" class and adds a "StudentMapper" class with stereotype «DataMapper». The designer uses the automatic structural repair provided by the PLA to complete the structure of this pattern, which results in adding operations for find, insert, delete, and update to the "StudentMapper" class, as well as adding a dependency to a supplier class named "SupplierCls" (see Figure 49).

Step 11: Add Patterns to the Data Source Layer The designer decides to use the Table Data Gateway pattern as the supplier for the Data Mapper. Therefore, he/she renames the "SupplierCls" of the Data Source Layer to "StudentTDG" and applies the stereotype «TableDataGateway» on this class. The designer leaves the details of completing



Please select the elements to be added to the class:

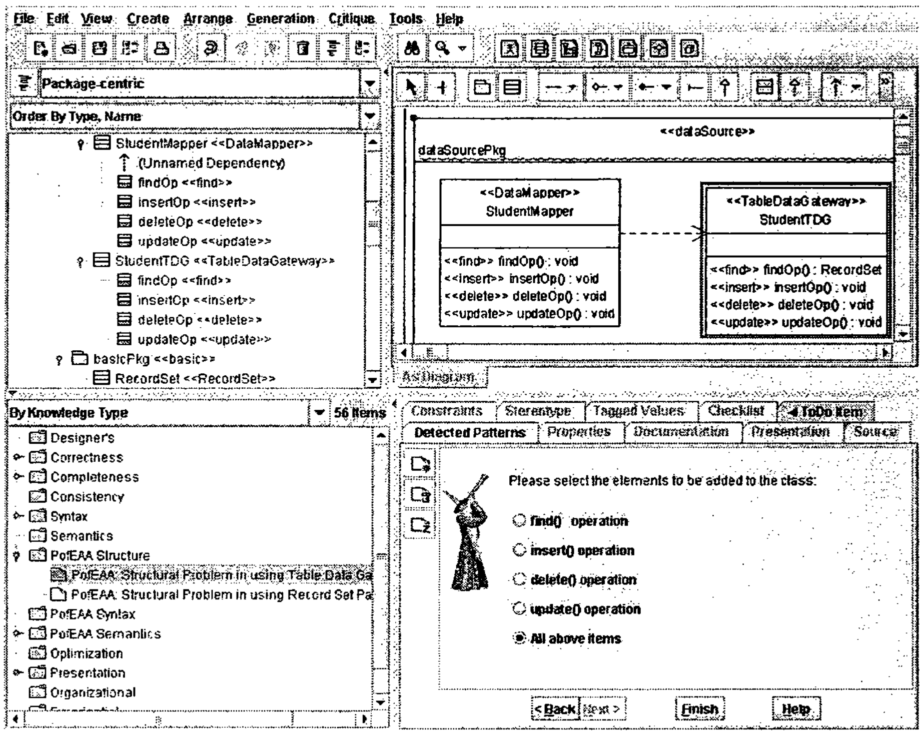
- find() operation
- insert() operation
- delete() operation
- update() operation
- supplier class
- [✓ All above items]

Figure 49: Automatic Fix of Structural Problems in Data Mapper Pattern

the structure of the Table Data Gateway pattern to the ArgoPLV. Upon the designer's request, the PLA completes the structure of the "StudentTDG" class by adding find, insert, delete, and update operations (see Figure 50).

Step 12: Cascaded Problem Repair adds a Pattern to the Base Layer Automatic instantiation of the Table Data Gateway pattern by the PLA in the previous step causes a new structural error (see the ToDo Pane in Figure 50). This is because, the PLA has also applied the syntactic rule that requires the type "Record Set" as the return type of the find operation in the Table Data Gateway pattern. Therefore, the return type of the added "find" operation is set to a type named "RecordSet." This type is added to the model as a class named "RecordSet" with stereotype «RecordSet» in the Base Layer of the model by the PLA (see the Explorer Pane in Figure 50), which then triggers a new structural error regarding the incorrect application of the Record Set pattern (see the ToDo Pane in Figure 51).

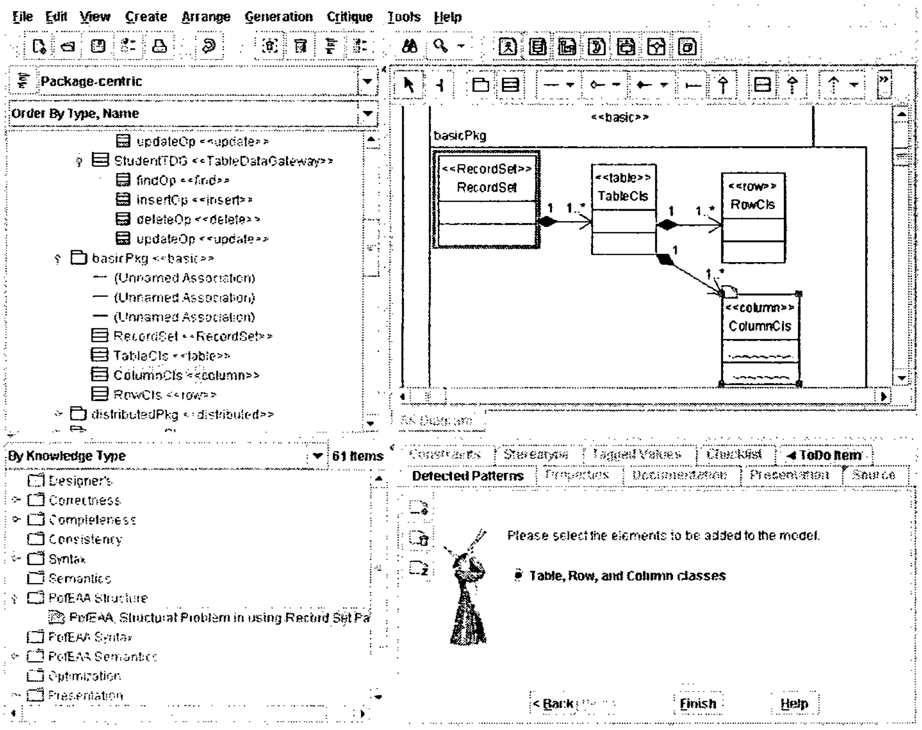
Again, the designer asks the PLA to fix the problems in the Record Set pattern which causes three classes for Table, Row, and Column to be created in the Basic Layer of the model. Also, the PLA builds the containment associations between the Record Set, Table, Row, and Column classes (see Figure 51).


File Edit View Create Arrange Generation Critique Tools Help
Package-centric
Order By Type, Name
StudentMapper <<DataMapper>>
(Unnamed Dependency)
findOp <<find>>
insertOp <<insert>>
deleteOp <<delete>>
updateOp <<update>>
StudentTDG <<TableDataGateway>>
findOp <<find>>
insertOp <<insert>>
deleteOp <<delete>>
updateOp <<update>>
basicPkg <<basic>>
RecordSet <<RecordSet>>
<<dataSource>>
dataSourcePkg
<<DataMapper>> StudentMapper
<<find>> findOp() : void
<<insert>> insertOp() : void
<<delete>> deleteOp() : void
<<update>> updateOp() : void
<<TableDataGateway>> StudentTDG
<<find>> findOp() : RecordSet
<<insert>> insertOp() : void
<<delete>> deleteOp() : void
<<update>> updateOp() : void
As Diagram
By Knowledge Type
56 Items
Designers
Correctness
Completeness
Consistency
Syntax
Semantics
PoEAA Structure
PoEAA: Structural Problem in using Table Data Ga
PoEAA: Structural Problem in using Record Set Pa
PoEAA Syntax
PoEAA Semantics
Optimization
Presentation
Organizational
Constraints Stereotype Tagged Values Checklist ToDo Item
Detected Patterns Properties Documentation Presentation Source
&lt; Back Next &gt; Finish Help

Please select the elements to be added to the class:

- ○ find() operation
- ○ insert() operation
- ○ delete() operation
- ○ update() operation
- [✓ All above items]

Figure 50: Automatic Fix of Structural Problems in Table Data Gateway Pattern



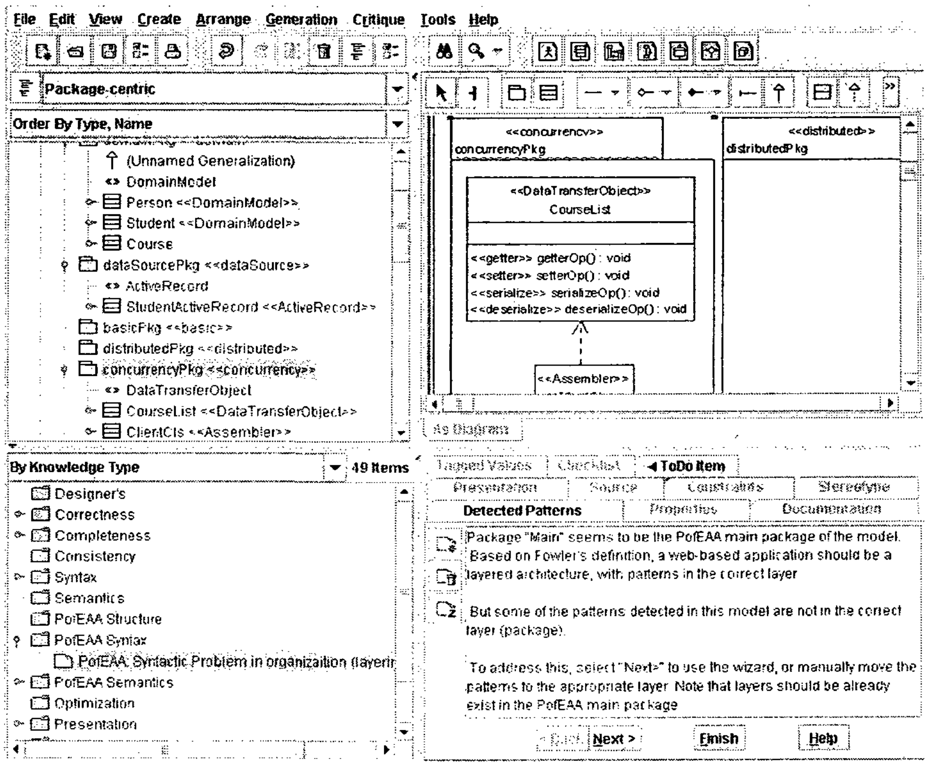
Please select the elements to be added to the model.

- [✓ Table, Row, and Column classes]

Figure 51: Automatic Fix of the Structural Problems in Record Set pattern

Step 13: Design the Distributed Layer The designer decides to use the Data Transfer Object pattern to provide "CourseList" which is a coarse-grained facade for accessing the list of courses taken by a student. The designer defines a "CourseList" class in the Concurrency Layer (instead of Distributed Layer) by mistake, then he/she applies the stereotype «DataTransferObject» on the "CourseList" class. He/she completes the structure of the pattern by adding all the required operations (getter, setter, serialize, and deserialize), along with a corresponding Assembler class (see the Editor Pane in Figure 52).

Syntactic Problem Detection Regarding the Organization of Patterns The PTV triggers a syntactic error due to the misplacement of the pattern Data Transfer Object (see the ToDo Pane in Figure 52). The PLA shows the guidelines about the problem, and gives the designer option to fix the problem automatically (see the Details Pane in Figure 52).



Package "Main" seems to be the PofEAA main package of the model. Based on Fowler's definition, a web-based application should be a layered architecture, with patterns in the correct layer

But some of the patterns detected in this model are not in the correct layer (package).

To address this, select "Next>" to use the wizard, or manually move the patterns to the appropriate layer. Note that layers should be already exist in the PofEAA main package

Figure 52: Reporting the Syntactic Problem Regarding Organization of Patterns

Syntactic Problem Repair Upon designer's request, the PLA moves the Data Transfer Object pattern to the Distributed Layer (see the Explorer Pane in Figure 53). However, note that the graphical view of the model (the Editor Pane) is not automatically refreshed by the PLA. It is up the designer to synchronize the graphical view of the model shown in the Editor Pane with the hierarchical view of the model shown in the Explorer Pane.

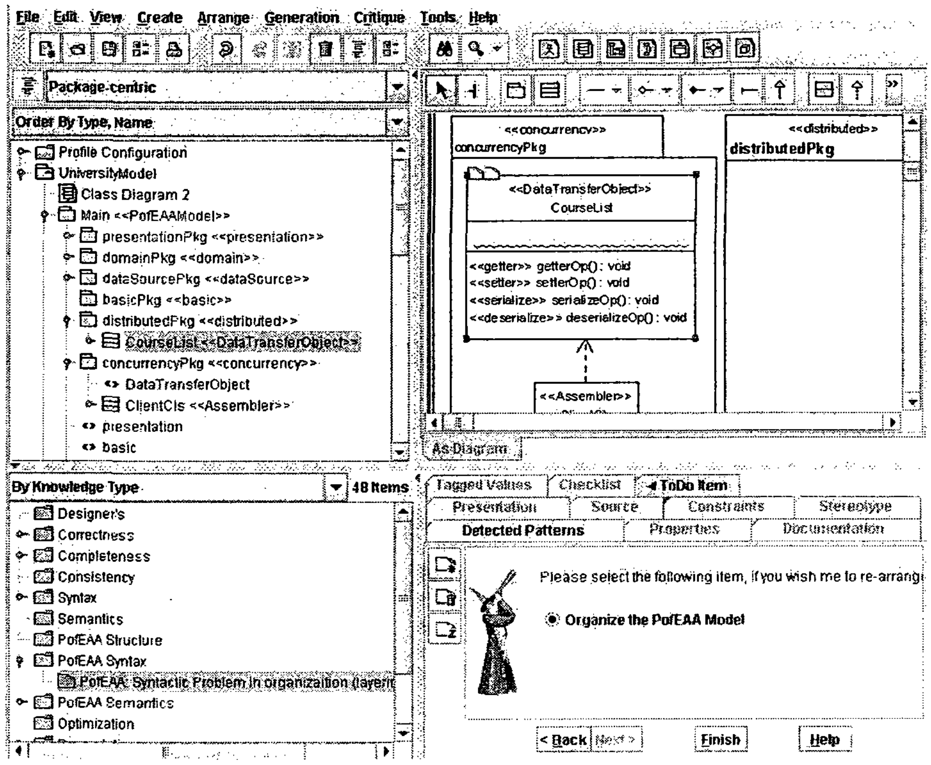
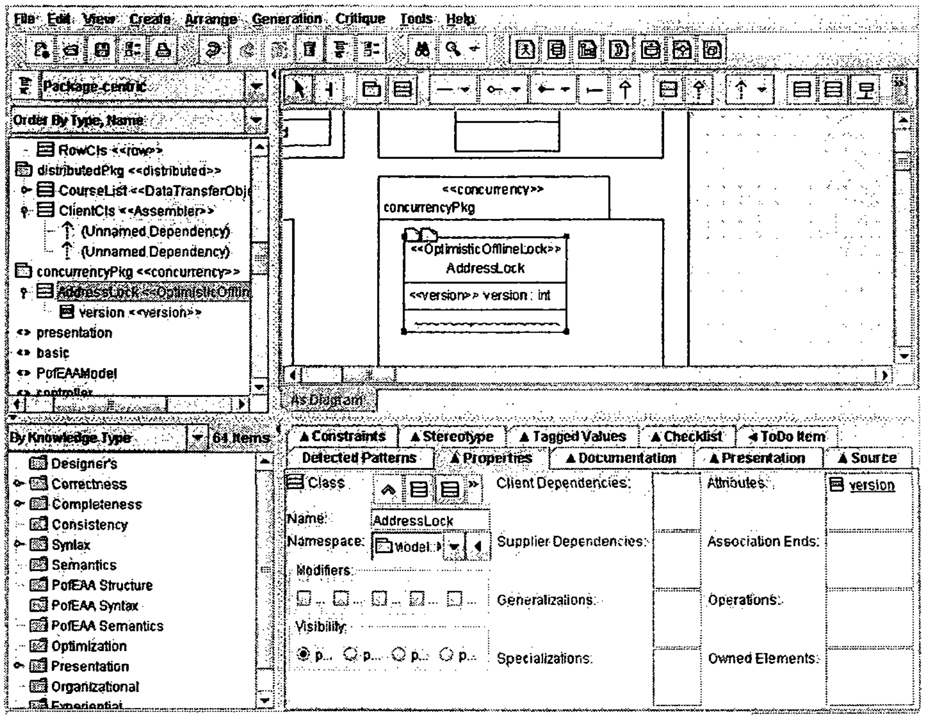


Figure 53: Automatic Reorganization of Patterns into the Layers

Step 14: Design the Concurrency Layer For preventing any conflict between the transactions that manipulate the Address class, the designer decides to apply the Optimistic Offline Lock pattern. This is an appropriate choice due to the fact that the possibility of conflicts is presumed to be low: Remember the setting of tagged value {ChanceOfConflict=Low} in step 4. The designer defines a class named "AddressLock" with stereotype «OptimisticOfflineLock» in the Concurrency Layer. He/she completes the structure of the pattern by adding the required "version" attribute to the "AddressLock" class (see the Editor Pane in Figure 54).

Step 15: Review the Design Model and the Applied Patterns The designer reviews the current state of the design of the system which has no structural, syntactic, or semantic errors, from the ArgoPLV point of view (see the Editor Pane in Figure 55). Also, the designer reviews the list of patterns applied so far in the design (see the Details Pane in Figure 55).



Name: AddressLock
Namespace: Model

Figure 54: Applying the Optimistic Offline Lock pattern in the Concurrency Layer

Step 16: Review the Design Rationale The designer reviews the modifications that are made by the PLA by reviewing the Design Rationale file. Table 17 represents some records extracted from the Design Rationale file regarding this design model.

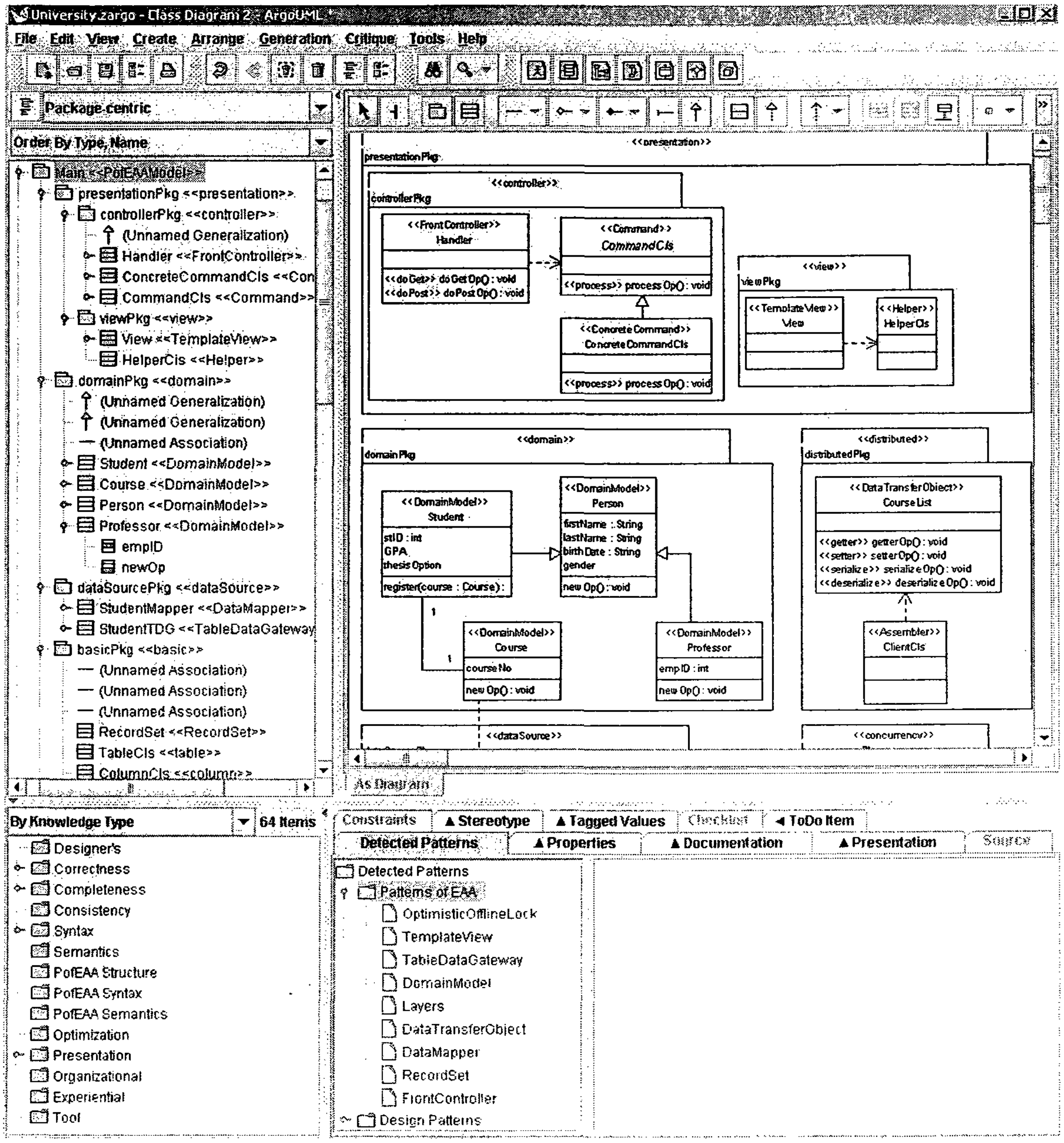


Figure 55: The Design Model for the Application and the Applied Patterns

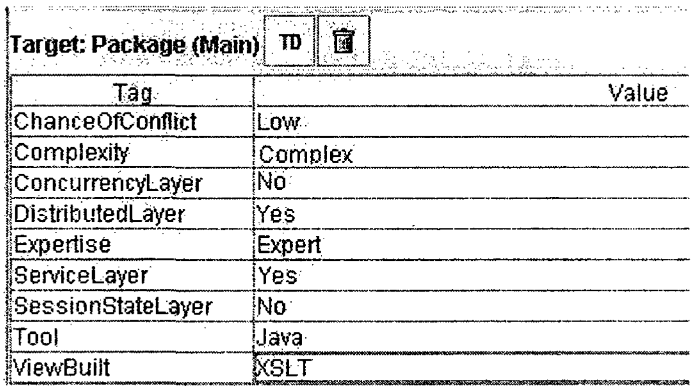
Table 17: Records from the Design Rationale File Associated with the Repairs

Date/Time	Wizard Class	Issue	Rationale
2009-04-08 11:20:32	WizLayers	PofEAA: Syntactic Problem - Missing Layers in the Model	A design built based upon the PofEAA patterns needs layers such as Presentation, Domain, and Data Source. Other Layers such as Service, Basic, Distributed, Concurrency, and Session State, depend upon the context information set by the tagged values. This wizard has added any of those missing items to the model.
2009-04-08 11:31:45	WizFrontController	PofEAA: Structural Problem in using Front Controller Pattern	The Front Controller pattern needs a "Handler" class with goGet and doPost operations as well as an Abstract Command class with a Process operation and at least one concrete child. This wizard has added any of those missing items to the model.
2009-04-08 12:12:11	WizTemplateView	PofEAA: Structural Problem in using Template View Pattern	The Template View pattern needs a supplier "Helper" class. This wizard has added any of those missing items to the model.
2009-04-08 12:39:25	WizViewLayerSem	PofEAA: Semantic Problem regarding the View Layer of the model	The patterns of the View Layer should match with the context information, especially with the value of ViewBuilt tag. This wizard has changed the tag correspondingly.
2009-04-08 16:41:30	WizDataMapper	PofEAA: Structural Problem in using Data Mapper Pattern	The Data Mapper pattern needs CRUD operations as well as a supplier class. Also the class should be stateless, i.e., has no public attribute. This wizard has added any of those missing items to the class. But it is up to the designer to make sure that the class is stateless.
2009-04-08 16:45:28	WizTableData Gateway	PofEAA: Structural Problem in using Table Data Gateway Pattern	The Table Data Gateway pattern needs CRUD operations. Also the return type of all the "find" operations should be Record Set. This wizard has added any of those missing items to the model. The Record Set is added if required.
2009-04-08 16:57:53	WizRecordSet	PofEAA: Structural Problem in using Record Set Pattern	The Record Set pattern needs aggregation association to Table, Row, and Column classes. This wizard has added any of those missing items to the model.
2009-04-08 18:34:27	WizPatterns	PofEAA: Syntactic Problem in organization (layering) of patterns	A design built based upon the PofEAA patterns needs to have each pattern in its corresponding layer.This wizard has rearranged model such that each pattern is placed in the appropriate layer.

5.3 Using ArgoPLV in Checking a Design Model of the Application

In this section, we show how the ArgoPLV tool helps a designer verify a model which is already built for the *Online Student Registration System* based upon the patterns of the *PofEAA* PL. We suppose that the model is saved in ".zargo" format which is an ArgoUML recognizable format. If the model is serialized in the XMI format, there is no graphical view for the model; However, the ArgoPLV is able to verify the model and give the errors as usual. Also, we suppose that the designer has utilized the stereotypes and the tagged values of the "*PofEAA* UML Profile" to specify the patterns that are applied in the model. Due to lack of space, the given model deals with the requirements of the system which are of student's interest. We show the verification process via a sequence of steps taken by the designer.

Step 1: Load the Model into ArgoPLV After loading the model, the context information can be investigated. Figure 56 shows the context information, i.e., the tagged values of the «PofEAA» main package.


Target: Package (Main)

Tag	Value
ChanceOfConflict	Low
Complexity	Complex
ConcurrencyLayer	No
DistributedLayer	Yes
Expertise	Expert
ServiceLayer	Yes
SessionStateLayer	No
Tool	Java
ViewBuilt	XSLT

Figure 56: The Tagged Values of the Main Package of the Model

Figure 57 shows the class diagram of the design loaded into ArgoPLV. Note that, we have shown the diagram as it is appeared in the Editing Pane, hence, the other ArgoUML Panes are not displayed in the figure.

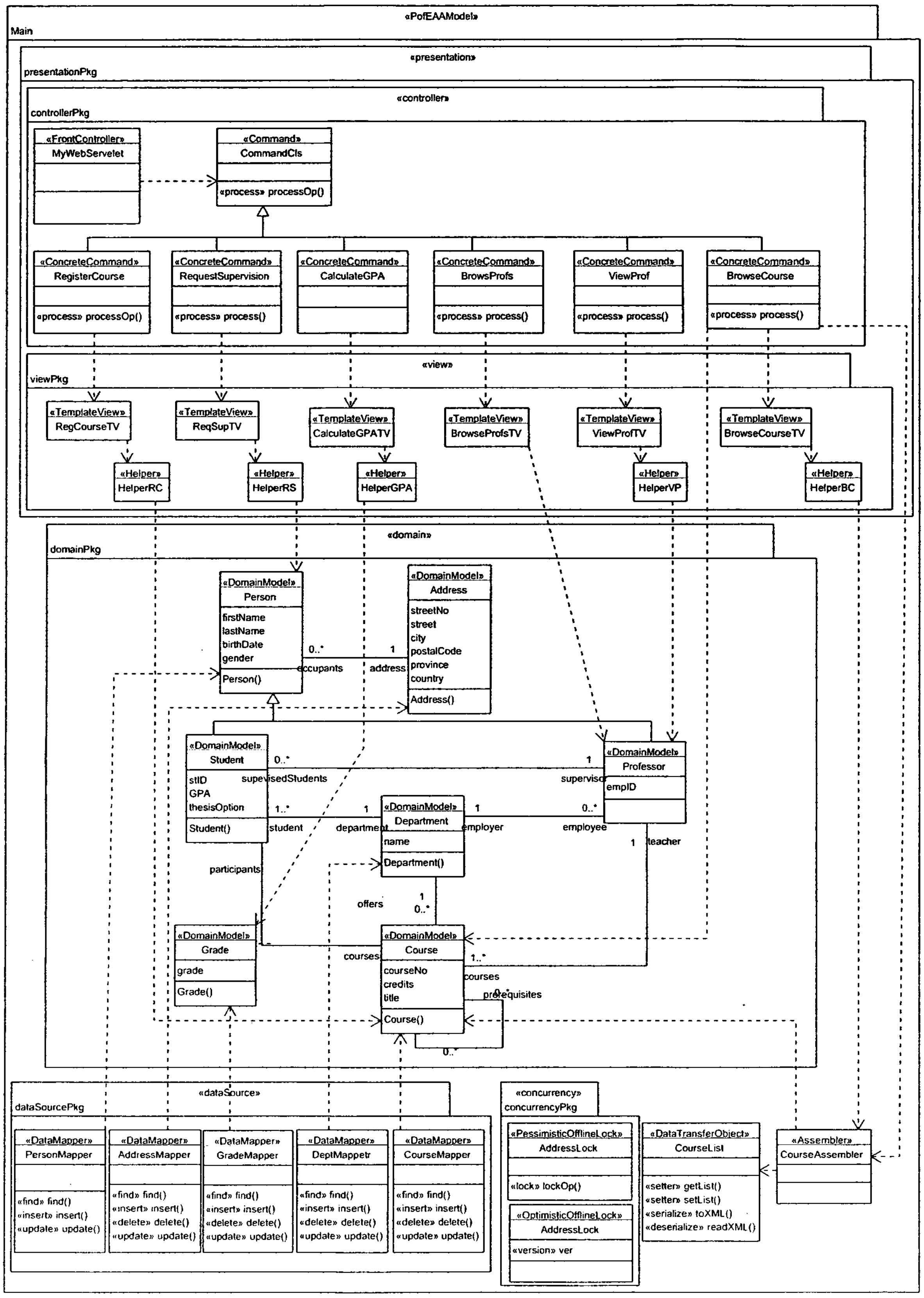


Figure 57: A Design Model for Online Student Registration System using *PofEAA* Patterns

Step 2: Check the Structural Problems of the Model Several structural errors are detected by the PSV and reported by the PLA. In the following, we elaborate on the errors and their causes. See Figure 57 while reviewing the list of errors.

1. Structural problem in using the Front Controller pattern: The causes of this problem are:
   (a) missing the "doGet" and "doPost" operations in the Handler class ("MyWeb-Servelet"),
   (b) having a non-abstract Command class ("CommandCls"), and
   (c) missing the "process" operation in one of the Concrete Command classes ("CalculateGPA").
2. Structural problem in using the Template View pattern: The cause of this problem is missing the Helper class for one of the Template View classes ("BrowseProfsTV").
3. Structural problem in using the Domain Model pattern: The cause of this problem is that one of the Domain Model classes ("Professor") has no operation.
4. Structural problem in using the Data Mapper pattern: The cause of this problem is missing the "delete" operation in one of the Data Mapper classes ("PersonMapper').

Step 3: Fix the Structural Problems The designer asks the PLA to fix the structural problems automatically. Therefore the following repairs, corresponding to the above errors, are applied to the model.

1. (a) A "doGetOp" operation and a "doPostOp" operation are added to the "MyWeb-Servelet" class.
   (b) The "Command" class is set as an abstract class.
   (c) A "processOp" operation is added to the "CalculateGPA" class.
2. A "HelperCls" class is created as a supplier for the "BrowseProfsTV" class.
3. A "newOp" operation is added to the "Professor" class.
4. A "deleteOp" operation is added to the "PersonMapper" class.

Step 4: Check the syntactic Problems of the Model The following list shows the syntactic errors detected by the PTV and reported by the PLA.

1. Syntactic problem in the layering of the model: The cause of this problem is the missing Distributed Layer in the model while the designer has shown his/her intention of having a Distributed Layer via the corresponding tagged value.

2. Syntactic problem in the organization of patterns inside layers: The cause of this problem is that the Data Transfer Object pattern ("CourseList" class) is not located in its corresponding package (Distributed Layer).

3. Syntactic problem in the Concurrency Layer: The cause of this problem is that there exist two conflicting patterns (two "AddressLock" classes) in the Concurrency package.

Step 5: Fix the Syntax Problems The designer asks for help from PLA. The following repairs, corresponding to the above errors, are applied to the model, either manually by the designer or automatically by the PLA.

1. The PLA creates a Distributed Layer ("distributedPkg" package) inside the Main package.

2. The PLA moves the "CourseList" class into the Distributed Layer ("distributedPkg" package).

3. The designer removes the "AddressLock" class (the class with stereotype «PessimisticOfflineLock») from the Concurrency package.

Step 6: Check the Semantic Problems The following list shows the semantic errors detected by the PMV in the given design.

1. Semantic Problem regarding the View Layer: The cause of this problem is that the tagged value "ViewBuilt=XSLT" is in contradiction with usage of the Template View pattern.

2. Semantic Problem regarding the Service Layer: The cause of this problem is that the designer has shown that he/she wants to have a Service Layer in his/her design by setting tagged value "ServiceLayer=Yes." but, there is no such layer in the model.

Step 7: Fix the Semantic Problems

1. The designer sets tagged value "ViewBuilt=HTML" via the text box provided by the PLA.
2. The designer decides not to have a Service Layer and sets tagged value "Service-Layer=NO" via the text box provided by the PLA.

Step 9: Final Design Figure 58 shows the package diagram of the design model after all the errors caught by the ArgoPLV are fixed as described in the above.

5.4 Discussion

5.4.1 Summary

To show the applicability and usefulness of the ArgoPLV, it is used in building a design model based upon the patterns of *PofEAA* for a sample application: *Online Student Registration System.* The application is a web-based system, consisting of students, professors, departments and courses, that handles the requests form both students and professors regarding the courses and supervisions. By two different cases, it is shown that the ArgoPLV is useful in both applying a single pattern and connecting the patterns together. The first case reveals the usefulness of ArgoPLV as a critiquing system that guides the designer in step-by-step design of the system based on the patterns of *PofEAA*. This case also shows how the PLA can help a novice designer in pattern instantiation, pattern layering, and pattern weaving. The second case indicates the power of Pattern Language Verifier (PLV) as an offline verifying process. The designer checks an existing design model, which is saved in an XMI file, with the ArgoPLV. ArgoPLV informs the designer of all the structural, syntactic, and semantic errors in the model.

5.4.2 Observations

Since ArgoPLV is a critiquing system, it is more attractive to be used in an interactive mode (like the first case). However. ArgoPLV is not intrusive, designer can totally ignore it.

It is easier for the designer to instantiate a pattern by the help of the PLA instead of referring the text sources to understand the correct structure of a pattern. The designers have to know the "Sign" of the pattern they want to use, before applying it, but not its full structure. The tool helps him/her in building the correct structure.

Verification of the pattern relationships, and guiding the designer in instantiating a pattern as a sequel of another pattern, is the structuring mechanism of the PL which is very helpful, especially for the novice designers. This is an important aspect of ArgoPLV, because only very expert designers are careful and aware about these relationships.

Semantic correctness of the model, i.e., consistency of the model with the context information, is another useful service of the ArgoPLV that ensures the designer about the consistency of the model.

Having a real time critiquing tool is similar to the idea of real time compilation, which exists in modern Integrated Development Environments (IDEs) such as Eclipse.

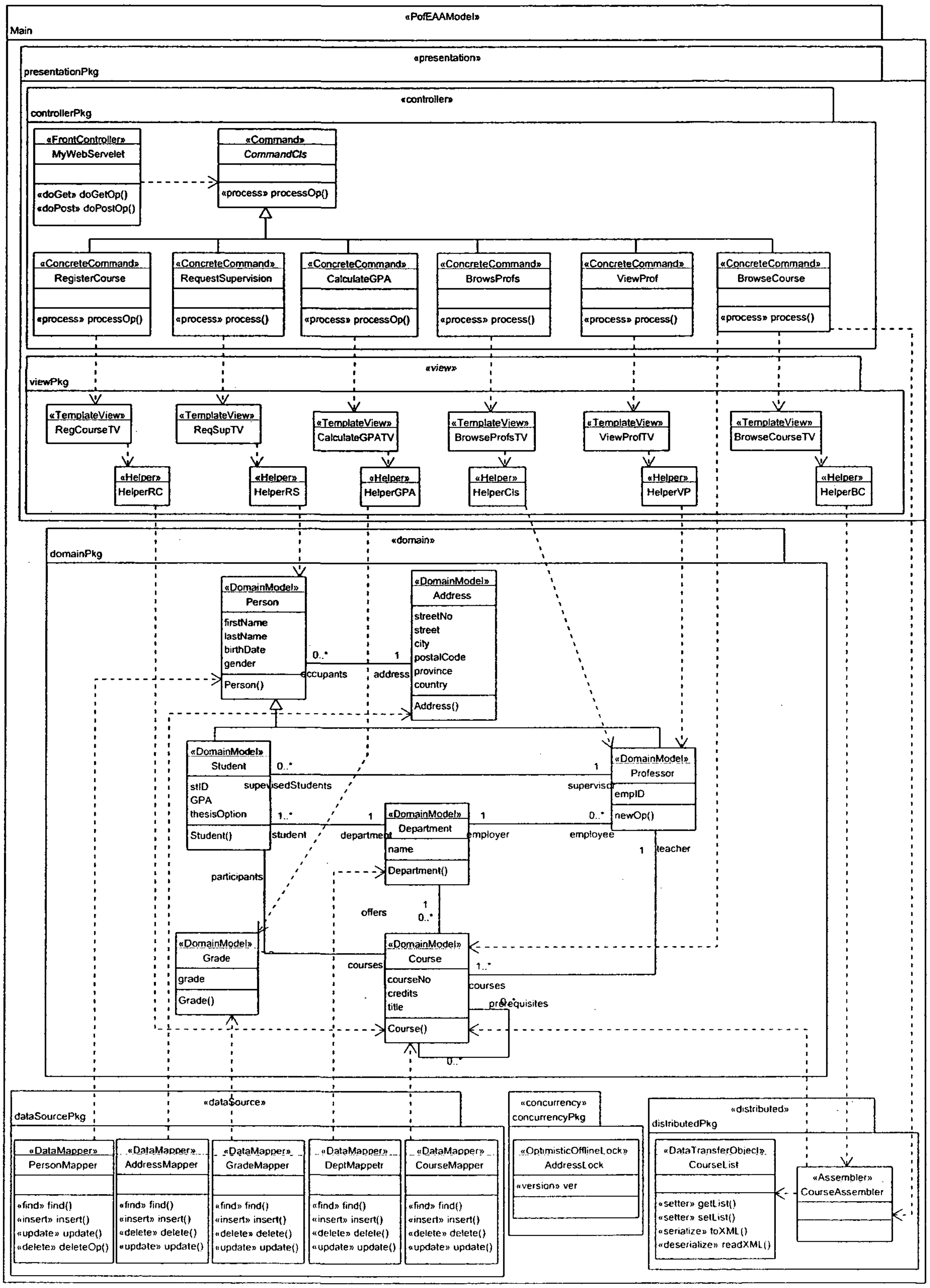


Figure 58: Design of Online Student Registration System - Refined by ArgoPLV

Chapter 6

Conclusion

6.1 Summary

In model-driven software development approaches, software designers are interested and are encouraged to apply patterns in their designs in the hope of generating better designs. A Pattern Language (PL) is a collection of inter-related patterns with a guiding rhythm that starts from one pattern and helps the designers on how to move from one pattern to another, such that at the end, the whole system is designed. *Designing with patterns* of a PL is not an easy task, especially for the novice designers.

In this thesis, we argued that building a design model based upon the patterns of a PL can be viewed as writing a program in a programming language. Borrowing the ideas from the compilers, we introduced a process named Pattern Language Verifier (PLV), and we elaborated that building a PLV for a given PL, requires the structural, syntactic, and semantic rules of the PL to be explicitly and precisely defined.

We presented three formalisms for defining these three groups of rules. Since we limited our work to UML models (class diagrams and package diagrams), we utilized the UML Profile mechanism to ease the pattern naming and the detection of pattern elements for the tool, as well as eliminating the problem of Pattern Selection from the scope of the work. Hence, we emphasized that the PLV is a profile-driven process, and to have a PLV for a PL, it is required that the profile for that PL be already defined.

As a case study, we selected a subset of Patterns of Enterprise Application Architecture (*PofEAA*) as a PL. We defined a UML profile for the selected PL. We extracted the advices from the *PofEAA* book, then we transformed those advices into the formal rules which are used by the PLV. We hand coded the rules (the profile constraints) into the ArgoUML modeling tool to obtain a PLV for *PofEAA*, which we called it the ArgoPLV.

To show how the ArgoPLV may help a designer in designing a system based upon the *PofEAA*, we designed a sample application, an *online student registration system.* We divided our case study into two parts. In the first part, we showed a step-by-step design of the application. We discussed how different kinds of errors are caught by the tool, and then it helps the designer in repairing errors. In the second part, we showed how the tool can be used in offline mode, like a compiler, to verify an existing model which is built for the application and to report the errors.

6.2 Review of the Contributions

We have made the following contributions:

1. The PLV process (See Section 3.5). This thesis moves the state-of-the-art in the Pattern Language Verification to the next level by introducing the PLV process. The work can be considered as an extension of the well-researched idea of "automatic pattern detection" to a broader idea called PLV, which focuses more on verifying the relationship between patterns. The work presented here is an improved version of our previously published ideas in [ZKB08, ZBK09]. The PLV process is influenced by the programming language compilers, and this makes it a novel idea. PLV verifies the use of a PL in a UML design model. In addition to analyzing the model, PLV is equipped with a module, called Pattern Language Advisor (PLA), for helping the designer fix the problems. The PLA per se is a step forward to the Model-Driven Engineering (MDE) promise of automatic code generation.

2. A formalism for representing a PL (See Section 3.3). While there exist attempts on formalizing the pattern relationships, none of the previous work has addressed all the *structural*, *syntactic*, and *semantic* aspects of a PL altogether. This thesis moves the state-of-the-art in pattern formalization techniques, because it addresses all these three aspects.

3. The ArgoPLV (See Chapter 4). As a proof of concept, a PLV for the *PofEAA* PL is built. This work itself has resulted in several useful artifacts and experiences:

    (a) The *PofEAA* Advices (See Section 4.2). Extracting the advices from the book and classifying them into three groups structural, syntactic, and semantic, is a useful source of knowledge (quick reference) for the designers who want to apply these patterns.

(b) The formalized *PofEAA* rules (See Section 4.2). The advices are formalized using the formalism proposed in this thesis. These formalized rules pave the way for defining the constraints of the profile. Further, these rules can be used as a compact and quick view of the PL, useful for more advanced designers.

(c) The *PofEAA* UML Profile (See Section 4.3). Many profiles have been introduced to the UML community. However, this is the first time that a profile is defined for a PL. The profile per se is a contribution of this thesis, since it can be used by both the designers and the researchers.

4. An exemplar session of ArgoPLV (See Chapter 5). This example shows *designing with patterns* for an application: *Online Student Registration System.* This also can be viewed as a walkthrough on applying the *PofEAA* PL in designing a system.

5. An MDE Road Map (See Section 2.1). People have discussed MDE from different aspect. We give our view of MDE as a road map, followed by a discussion on the artifacts, the transformations, the modeling tool, and the issue of "Quality in Modeling."

6.3 Discussion

The detailed discussion about the PLV, ArgoPLV, and the application of ArgoPLV in action, are already given at the end of the corresponding chapters, Chapter 3, Chapter 4, and Chapter 5. In the following we mention more general issues we encountered during this research.

***PofEAA* as a PL** PL, as defined by Alexander [A$^+$77] and adapted in this thesis, is not a mature concept in software yet. Most of the existing pattern collections are only a catalog of patterns. Only a few of these collections fulfill the definition of PL. Indeed, this is true for the *PofEAA*, considering the Fowler's confession: "Certainly none of my books have been pattern languages" [Fow06]. Although *PofEAA* per se is not a PL, we selected a subset of its patterns, and extracted a set of coherent advices from *PofEAA* such that the result is very close to our definition of PL.

Subset of *PofEAA* *PofEAA* is full of advices, suggestions. tradeoffs, and even story tellings in terms of alternative solutions for a problem. Therefore, extracting advices from the *PofEAA* book, which is not too close to the standards of a PL, is not straightforward. An expert must do this task in order not to include inconsistent advices. Translating the advices into the formal rules also needs expertise, and sometimes, needs interpretations. In some cases, maybe the expert wants to enforce his/her idea and modify some of the pattern definitions. This is possible, but care must be taken to announce that the result is maybe a new PL.

Profile Constraints While hard-coding the profile constraints into a modeling tool (e.g., ArgoUML) gives more flexibility to the programmer than what the OCL offers, it is cumbersome. Especially, duplicate coding is required in ArgoUML *wizards* to make sure that the criteria that have triggered the *critic* are still valid.

OCL OCL is assumed to be the companion of UML for writing constraints. However, there are limitations reported for OCL that must be addressed, especially for supporting the emerging model-driven paradigms [CDGW06]. For instance, ambiguities in OCL must be fixed, good support for OCL in Eclipse framework must be provided, and efficiency of evaluating OCL constraints must be improved [MLC06]. "OCL is hard to understand and, as a consequence, difficult to use" [CBC05], and we believe that working with OCL is still not comfortable for people. The evidence is the emergence of OCL-like languages such as EOL [KPP06].

Pattern Relationships Formalizing and characterizing the relationships between patterns is *still* an open research problem. The reason is that the patterns are not 100% static elements like keywords in a programming language. A pattern is a compound element (recall the sections of a *pattern form*), therefore, it is not easy to define the relationship between one pattern to another.

ArgoUML Selecting ArgoUML as the platform for implementing the PLV for *PofEAA* has both pros and cons. The pros are: ArgoUML is a design critiquing system. Hence, for providing interactive support to the designer, ArgoUML is an appropriate tool. ArgoUML is an open source tool with more than 500000 downloads during last decade. ArgoUML is under upgrade by a team of experienced developers. The cons are: ArgoUML does not have powerful support for OCL, particularly, there is no support for writing OCL at meta-model level. ArgoUML has no determined plan for supporting UML 2.0 in future.

6.4 Limitations

There are some deficiencies in our work, due to the originality of the idea and limited time, which are summarized as following.

The PLV Process The PLV is a profile-driven process. This will limit the scope of the applicability of the process, since the designers have to use the profile stereotypes and tagged values in their designs. Further, the PLV supposes that the designer knows which pattern he/she decides to apply, hence, does not provide any help in pattern selection. The PLV does not consider the domain of the underlying PL or the domain of the system under design. The PLV does not have precise definition of Semantic aspects of the design which are verified by the Pattern Language Semantic Verifier (PMV) module. Considering the inconsistencies between the context information with the use of patterns is a naive approach to look at the semantic issues.

The ArgoPLV Case Study The case study does not include all the patterns from *PofEAA*. Especially eliminating the Object-Relational Mapping patterns makes ArgoPLV a limited version rather than a tool which is usable in a real application. The extracted rules investigate only the static view of the design, more specifically, they only consider the class diagram and the package diagram. This will prevent us from detecting patterns that are more about the implementation techniques, e.g., the Pessimistic Offline Lock pattern, effectively. In designing a system, there are several points to start, while the syntactic rules defined for ArgoPLV select only one point as the start pattern. This is a limitation for the designer.

The Application Design Case Study Testing ArgoPLV with a small application (*on-line student registration system*) is not sufficient to validate the tool. The system is not complicated enough to show the problem of selecting a pattern amongst a number of alternatives. The test is performed by builder of the ArgoPLV tool, hence, it does not exactly resemble a case in the real-world.

6.5 Comparison to Related Work

In this section we compare the PLV (and ArgoPLV) with the related work which are introduced in Chapter 2.

Pattern Enforcing Compiler (PEC) The most related work to PLV is the PEC [LSV05] (See Section 2.5.2). PEC uses a naming convention for easing the detection of class features, and uses interfaces as markers for showing the developer's desire for applying a pattern. This is similar to the usage of stereotypes in the PLV. PEC uses Javadoc to document pattern usages, while ArgoPLV creates a Design Rationale. PEC is written for Java language, while ArgoPLV checks UML design models. PEC investigates only individual patterns; It does not consider PL issues. PEC shows only "pass" or "fail" message to the developer; There is no advisory system. PEC deals only with GOF design patterns, however, it is extensible, i.e., the user can define new patterns without requiring any new syntax for the Java language.

Systematic pattern selection using pattern language grammars and design space analysis This work [Zdu07], is also very close to our work (See Section 2.5.3). Indeed, we see our work has been recognized by Zdun as his future work. In his conclusion, Zdun says: "We envision further application areas for the approach; for instance, the pattern language grammars and design spaces can potentially be used as an input for model-driven tools" [Zdu07]. Zdun's work deals with architectural patterns as well as GOF design patterns. The main difference between the PLV and Zdun's work is that his work is not a verifying approach; it is a pattern selection mechanism. Also Zdun's work does not address the models directly. The overlap of our work with Zdun's work is that both use the grammar idea to formalize the relationship between patterns. However, Zdun annotates the grammar with the design qualities. The advantage of the Zdun's work is that it addresses inter-collection issues. That means, the design can be built by applying patterns from several PLs. Also his work considers the domain-specific design decisions.

Pattern Detection Tools There are several works on detecting a pattern in a design model or source code (See Section 2.5.1). Our work differs from these work since they only focus on individual patterns, and do not address the relationship between patterns. Also most of these works fall into the category of GOF design pattern detection. Interactive DEsign Assistant (IDEA) [BP02] and Design Pattern Detection Using Similarity Scoring [TCSH06] are more closer to our work than others. The IDEA is also integrated into the ArgoUML. However, the IDEA is only capable to detect 11 GOF patterns. The advantages of IDEA is that it considers both class diagram and collaboration diagram. The latter work is unable to detect four GOF patterns. However, the methodology is general and can be applied for any pattern collection.

Model Inconsistency Detection Tools Rule-Based Inconsistency Detection Engine (RIDE) [LEM02] (See Section 2.1.5) is close to our work from the viewpoint that it aims to find and repair the inconsistencies in the UML models . The model under investigation must first be converted to a production system representation. Then, RIDE uses JESS to execute production rules. The advantage of RIDE is that it is general and extensible; it is not limited to the pattern misuses.

6.6 Future Work

There are several paths to extend and improve the work presented in this thesis. Generalizing the idea of PLV and the experiences gained in this work towards a framework for "Pattern Language Verification" would result in a valuable contribution to patterns and PLs.

The PLV process can also be enriched with the idea of systematic pattern selection presented by Zdun [Zdu07]. The PLV idea should be broaden to cover the inter-collection pattern applications, e.g., verification of the relationship between patterns from different PLs. Considering dynamic models, e.g., sequence diagram, in addition to the static views of the design is also of great help in verifying behavioral patterns.

While people are studying (and working on) existing PLs, including pattern catalogs and pattern collections, working on formalism of patterns and PLs is a real need, especially, if we look for more help from the CASE tools. Furthermore, consolidating the different formalisms that are proposed for defining the rules of a PL would be a fruitful research. Having precise formalisms, we can investigate the possibility of automatic building of the PLV modules, similar to the idea of automatic scanner and parser generators (e.g., Lex & Yacc [LMB92]) in the compiler design. More advanced research would be adding a module to PLV for optimization (refactoring [Fow99]) of designs.

The PLV process is mimicking the analysis part of a compiler. Investigating the synthesis (code-generation) part of a compiler, may leads to a research which consolidates the PLV with the MDSE approaches that promote full code generation from the UML models, such as xUML [MB02].

As simpler but more applied track for future work is to apply the PLV process (maybe modified version) for more PLs. and to build verifier tools that help designers in *designing with patterns.* Due to widespread usage of Eclipse, building the PLVs as Eclipse plugins has better chance of popularity.

Bibliography

[A$^+$77] Christopher Alexander et al. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[Ale79] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[Amb02] Scott Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process.* John Wiley & Sons, first edition, 2002.

[AN04] Jim Arlow and Ila Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML.* Addison-Wesley, 2004.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[B$^+$09] Kent Beck et al. Manifesto for agile software development. `http://www.agilemanifesto.org/`, [July 1, 2009].

[BBS05] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of design patterns in Java. In David F. Redmiles et al., editors, *ASE*, pages 224–232. ACM, 2005.

[BC09] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. Technical Report No. CR-87-43, Presented at OOPSLA'87, Online at `http://c2.com/doc/oopsla87.html`, [July 1, 2009].

[BCO05] Ruth Breu and Joanna Chimiak-Opoka. Towards systematic model assessment. In Jacky Akoka et al., editors, *ER (Workshops)*, volume 3770 of *Lecture Notes in Computer Science*, pages 398–409. Springer, 2005.

[Ber94] Steve Berczuk. Finding solutions through pattern languages. *Computer*, 27(12):75–76, Dec. 1994.

[Béz06] Jean Bézivin. Model driven engineering: An emerging technical space. In Ralf Lämmel et al., editors, *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer, 2006.

[BHS07a] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, volume 4. John Wiley & Sons, 2007.

[BHS07b] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*, volume 5. John Wiley & Sons, 2007.

[BIJ06] Alan W. Brown, Sridhar Iyengar, and Simon Johnston. A Rational approach to model-driven development. *IBM Systems Journal*, 45(3):463–480, Jul. 2006.

[BJ06] Jean Bézivin and Frédéric Jouault. Using ATL for checking models. *Electr. Notes Theor. Comput. Sci.*, 152:69–81, Mar. 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[Ble06] Alex Blewitt. *HEDGEHOG: Automatic Verification of Design Patterns in Java.* PhD thesis, University of Edinburgh, UK, 2006.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1. John Wiley & Sons, 1996.

[Boo09] Grady Booch. Handbook of software architecture. `http://www.handbookofsoftwarearchitecture.com/`, [July 1, 2009].

[BP02] Federico Bergenti and Agostino Poggi. Improving UML designs using automatic design pattern detection. In Shi-Kuo Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, volume 2, pages 771–784. World Scientific Publishing, 2002.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[CBC05] Dan Chiorean, Maria Bortes, and Dyan Corutiu. Proposals for a widespread use of OCL. In Thomas Baar, editor, *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005*, Technical Report LGL-REPORT-2005-001, pages 68–82. EPFL, 2005.

[CDGW06] Dan Chiorean, Birgit Demuth, Martin Gogolla, and Jos Warmer. OCL for (meta-)models in multiple application domains. In Thomas Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 152–158. Springer, 2006.

[Coc06] Alistair Cockburn. *Agile Software Development: The Cooperative Game.* Addison-Wesley, second edition, 2006.

[CPC+04] Dan Chiorean, Mihai Pasca, Adrian Cârcu, Cristian Botiza, and Sorin Moldovan. Ensuring UML models consistency using the OCL environment. *Electr. Notes Theor. Comput. Sci.*, 102:99 – 110, Nov. 2004. Proceedings of the Workshop, OCL 2.0 - Industry Standard or Scientific Playground?

[Dog07] Asif Dogar. Model Driven Development for Enterprise Applications. Master's thesis, Concordia University, 2007.

[DSM09] DSMForum. Domain Specific Modeling (DSM). `http://www.dsmforum.org/`, [July 1, 2009].

[Egy07] Alexander Egyed. Fixing inconsistencies in UML design models. In *ICSE*, pages 292–301. IEEE Computer Society Press, 2007.

[Fou09a] Eclipse Foundation. Eclipse Modeling Framework (EMF). `http://www.eclipse.org/emf/`, [July 1, 2009].

[Fou09b] Eclipse Foundation. Eclipse open source community. `http://www.eclipse.org/`, [July 1, 2009].

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, first edition, 2002.

[Fow03] Martin Fowler. Patterns [software patterns]. *Software, IEEE*, 20(2):56–57, Mar./Apr. 2003.

[Fow05] Martin Fowler. Language workbenches and model driven architecture. `http://www.martinfowler.com/articles/mdaLanguageWorkbench.html`, Jun. 2005. [July 1, 2009].

[Fow06] Martin Fowler. Writing software patterns. `http://www.martinfowler.com/articles/writingPatterns.html`, Aug. 2006. [July 1, 2009].

[FP97] Norman Fenton and Shari Lawrence Pfleeger. *Software metrics: a Rigorous and Practical Approach.* PWS Publishing Co., second edition, 1997.

[FQL+03] José M. Fuentes, Víctor Quintana, Juan Llorens, Gonzalo Génova, and Rubén Prieto-Díaz. Errors in the UML metamodel? *SIGSOFT Softw. Eng. Notes*, 28(6), Nov. 2003.

[FW90] Daniel P. Freedman and Gerald M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products.* Dorset House Publishing Company, third edition, 1990.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GSJ00] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise modeling of design patterns. In Andy Evans et al., editors, *UML*, volume 1939 of *Lecture Notes in Computer Science*, pages 482–496. Springer, 2000.

[HAZ07] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. Using patterns to capture architectural decisions. *IEEE Software*, 24(4):38–45, Jul./Aug. 2007.

[HC07] Scott Henninger and Victor Corrêa. Software pattern communities: Current practices and challenges. In *PLoP '07: Proceedings of the 2007 conference on Pattern languages of programs*, 2007.

[Hil09a] Hillside.net. Pattern languages of programs (PLoP) conference official web site. `http://hillside.net/plop/`, [July 1, 2009].

[Hil09b] Hillside.net. Patterns web site. `http://hillside.net/`, [July 1, 2009].

[IBM09a] IBM. Rational Rose. `http://www-01.ibm.com/software/awdtools/developer/rose/`, [July 1, 2009].

[IBM09b] IBM. Rational Software Architect (RSA). `http://www-306.ibm.com/software/awdtools/architect/swarchitect/`, [July 1, 2009].

[Int98] International Organization for Standardization (ISO). Information technology - software product quality. ISO9126, Part i: Quality Model Edition, 1998.

[JB06] Frédéric Jouault and Jean Bézivin. KM3: A DSL for metamodel specification. In Roberto Gorrieri and Heike Wehrheim, editors, *FMOODS*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006.

[Ken02] Stuart Kent. Model driven engineering. In Michael J. Butler et al., editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002.

[KJ04] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management*, volume 3. John Wiley & Sons, 2004.

[KKL$^+$98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, Jan. 1998.

[Knu64] Donald E. Knuth. Backus normal form vs. Backus Naur form. *Commun. ACM*, 7(12):735–736, Dec. 1964.

[Kob04] Cris Kobryn. UML 3.0 and the future of modeling. *Software and Systems Modeling*, 3(1):4–8, Feb. 2004.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon object language (EOL). In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006.

[KZ07] Holger Kampffmeyer and Steffen Zschaler. Finding the pattern you need: The design pattern intent ontology. In Gregor Engels et al., editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2007.

[Lab09] Sandia National Laboratories. JESS: the rule engine for the Java platform. `http://www.jessrules.com`, [July 1, 2009].

[Lan06] Christian F. J. Lange. Improving the quality of UML models in practice. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 993–996, New York, NY, USA, May 2006. ACM.

[Lar05] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall, third edition, 2005.

[LEM02] WenQian Liu, Steve Easterbrook, and John Mylopoulos. Rule-based detection of inconsistency in UML models. In *Workshop on Consistency Problems in UML-Based Software Development*, pages 106–123, Dresden, Germany, Oct. 2002.

[Lin06] Peter Linz. *An Introduction to Formal Languages and Automata.* Jones and Bartlett, 2006.

[LK05] Beate List and Birgit Korherr. A UML 2 profile for business process modelling. In Jacky Akoka et al., editors, *ER (Workshops)*, volume 3770 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2005.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc.* O'Reilly & Associates, 1992.

[LNH06] Daniel Leroux, Martin Nally, and Kenneth Hussey. Rational Software Architect: A tool for domain-specific modeling. *IBM Systems Journal*, 45(3):555–568, 2006.

[Lov06] Howard C. Lovatt. *A Pattern Enforcing Compiler (PEC) For Java.* PhD thesis, Macquarie University, Australia, 2006. Online at `https://pec.dev.java.net/nonav/introduction/index.html`.

[LSV05] Howard C. Lovatt, Anthony M. Sloane, and Dominic R. Verity. A pattern enforcing compiler (PEC) for Java: Using the compiler. In Sven Hartmann and Markus Stumptner, editors, *Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005)*, volume 43 of *CRPIT*, pages 69–78, Newcastle, Australia, 2005. ACS.

[MB02] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture.* Addison-Wesley, first edition, 2002.

[MCL04] Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun. Precise modeling of design patterns in UML. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society Press.

[MD97] Gerard Meszaros and Jim Doble. A pattern language for pattern writing. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design*, volume 3, pages 529–574. Addison-Wesley (Software Patterns Series), 1997.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[Mic09] Sun Microsystems. Metadata Repository (MDR). `http://mdr.netbeans.org/`, [July 1, 2009].

[MLC06] Gergely Mezei, Tihamér Levendovszky, and Hassan Charaf. Restrictions for OCL constraint optimization algorithms. *Electronic Communications of the EASST*, 5:1–18, Dec. 2006.

[MVN06] Dragos Manolescu, Markus Völter, and James Noble. *Pattern Languages of Program Design 5.* Addison-Wesley, 2006.

[NB02] James Noble and Robert Biddle. Patterns as signs. In Boris Magnusson, editor, *ECOOP*, volume 2374 of *Lecture Notes in Computer Science*, pages 368–391. Springer, 2002.

[Nob98a] James Noble. Classifying relationships between object-oriented design patterns. In *Software Engineering Conference, 1998. Proceedings. 1998 Australian*, pages 98–107, Nov. 1998.

[Nob98b] James Noble. Towards a pattern language for object oriented design. In *Technology of Object-Oriented Languages, 1998. TOOLS 28. Proceedings*, pages 2–13. IEEE Computer Society Press, Nov. 1998.

[Obj01] Object Management Group (OMG). Unified Modeling Language (UML): Specification, v1.4. OMG document: formal/01-09-67, 2001.

[Obj05a] Object Management Group (OMG). MOF 2.0/XMI mapping specification, v2.1. OMG document: formal/2005-09-01, 2005.

[Obj05b] Object Management Group (OMG). Unified Modeling Language (UML): Infrastructure, v2.0. OMG document: formal/05-07-05, 2005.

[Obj05c] Object Management Group (OMG). Unified Modeling Language (UML): Superstructure, v2.0. OMG document: formal/05-07-04, 2005.

[Obj06a] Object Management Group (OMG). Meta Object Facility (MOF): Core specification, v2.0. OMG document formal/2006-01-01, 2006.

[Obj06b] Object Management Group (OMG). Object Constraint Language (OCL): Specification, v2.0. OMG document: formal/06-05-01, 2006.

[Obj09a] Object Management Group (OMG). Catalog of UML profile specifications. `http://www.omg.org/technology/documents/profile_catalog.htm`, [July 1, 2009].

[Obj09b] Object Management Group (OMG). Model Driven Architecture (MDA). `http://www.omg.org/mda/`, [July 1, 2009].

[OOP09] OOPSLA. Object-oriented programming, systems, languages, and applications (OOPSLA) conference official web site. `http://www.oopsla.org/`, [July 1, 2009].

[PB88] Colin Potts and Glenn Bruns. Recording the reasons for design decisions. In *ICSE '88: Proceedings of the 10th International Conference on Software Engineering*, pages 418–427, Los Alamitos, CA, USA, Apr. 1988. IEEE Computer Society Press.

[PK02] Risto Pohjonen and Steven Kelly. Domain-specific modeling: Improving productivity and time to market. *Dr. Dobb's Journal*, Aug. 2002.

[Rob99] Jason E. Robbins. *Cognitive Support Features for Software Development Tools.* PhD thesis, University of California, Irvine, 1999.

[RR98] Jason E. Robbins and David F. Redmiles. Software architecture critics in the Argo design environment. *Knowledge-Based Systems*, 11(1):47 – 60, Sep. 1998.

[Rub90] Rubén Prieto-Díaz. Domain analysis: An introduction. *SIGSOFT Softw. Eng. Notes*, 15(2):47–54, Apr. 1990.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, Feb. 2006.

[sdm09] sdmetrics.com. SD-Metrics official web site. `http://www.sdmetrics.com/`, [July 1, 2009].

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, Sep. 2003.

[Sel06] Bran Selic. Model-driven development: Its essence and opportunities. In *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 313–319, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[Sel07] Bran Selic. A systematic approach to domain-specific language design using UML. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 2–9, Los Alamitos, CA, USA, 2007. IEEE Computer Society Press.

[SFJ96] Douglas C. Schmidt, Mohamed Fayad, and Ralph E. Johnson. Software patterns. *Commun. ACM*, 39(10):37–39, Oct. 1996.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice Hall, second edition, 1992.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2000.

[SV06] Tom Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

[Sys09] SysML. Systems Modeling Language (SysML). http://www.sysml.org/, [July 1, 2009].

[Tai07] Toufik Taibi. *Design Patterns Formalization Techniques.* IGI Publishing, 2007.

[TCSH06] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Software Eng.*, 32(11):896–909, Nov. 2006.

[The09] The Technische Universität Dresden. Dresden OCL toolkit. http://dresden-ocl.sourceforge.net/, [July 1, 2009].

[Tig09a] Tigris.org. ArgoUML official web site. http://argouml.tigris.org/, [July 1, 2009].

[Tig09b] Tigris.org. Gef official web site. http://gef.tigris.org/, [July 1, 2009].

[Unh05] Bhuvan Unhelkar. *Verification and Validation for Quality of UML 2.0 Models.* John Wiley & Sons, 2005.

[Uni09a] BABES-BOLYAI University. OCLE official web site. http://lci.cs.ubbcluj.ro/ocle/, [July 1, 2009].

[Uni09b] University of Nebraska Lincoln. Semantic Framework for Patterns (SFP). http://cse-ferg41.unl.edu/SFP/wiki/Main, [July 1, 2009].

[WdKqY$^+$03] Liu Wu-dong, He Ke-qing, Yingshi, Xu Hui, and Jiang Yi-xing. A pattern language model for framework development. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 669–673, Nov. 2003.

[Wir71] Niklaus Wirth. The design of a Pascal compiler. *Software Practice & Experience*, 1(4):309–333, Jul. 1971.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of the Technology of Object-Oriented Languages (TOOLS 26) Conference USA 1998*, pages 112–124. IEEE Computer Society Press, 1998.

[ZB07] Bahman Zamani and Greg Butler. Critiquing the application of pattern languages on UML models. In *Workshop on Quality in Modeling, MODELS2007 Conference*, pages 18–35, Nashville, TN, USA, 2007.

[ZBK09] Bahman Zamani, Greg Butler, and Sahar Kayhani. Tool support for pattern selection and use. *Electr. Notes Theor. Comput. Sci.*, 233:127–142, Mar. 2009. Proceedings of the International Workshop on Software Quality and Maintainability (SQM 2008).

[Zdu07] Uwe Zdun. Systematic pattern selection using pattern language grammars and design space analysis. *Software Practice & Experience*, 37(9):983–1016, Jul. 2007.

[ZHJ03] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a UML profile for software product lines. In Frank van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 129–139. Springer, 2003.

[Zim95] Walter Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1995.

[ZKB08] Bahman Zamani, Sahar Kayhani, and Greg Butler. A pattern language verifier for web-based enterprise applications. In Krzysztof Czarnecki et al., editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 553–567. Springer, 2008.

Appendix A

ArgoPLV Artifacts

In this appendix, we will present the artifacts that are produced during the process of building a Pattern Language Verifier (PLV) for the selected subset of Patterns of Enterprise Application Architecture (*PofEAA*) Pattern Language (PL). This appendix is organized into six sections:

1. Section A.1 shows the selected subset of *PofEAA* in a layered architecture.
2. Section A.2 shows the raw advices that are extracted from the *PofEAA* book. These advices are the base for defining the Structural, Syntactic, and Semantic rules of the PLV process.
3. Section A.3 shows the result of formalizing the advices into the formal rules using the formalisms defined in Chapter 3.
4. Section A.4 shows the different parts (Stereotypes, Tagged Values, and Constraints) of the *PofEAA* UML Profile.
5. Section A.5 shows the source code excerpts that clarify how the critics and wizards are hard coded in ArgoUML to define the constraints of the profile. Due to the importance of the class `GU`, its code is shown completely.
6. Section A.6 shows an example system, *Online Student Registration System*, which is designed based upon the *PofEAA*. Two versions of the design, both before and after verifying by ArgoPLV, are shown.
7. Section A.7 shows an excerpt of the Design Rationale file which is created during the verification of the design using ArgoPLV.

A.1 Selected Patterns from *PofEAA*

Figure 59 shows the subset of *PofEAA* that we have selected as our case study. This subset contains 23 patterns from several layers.

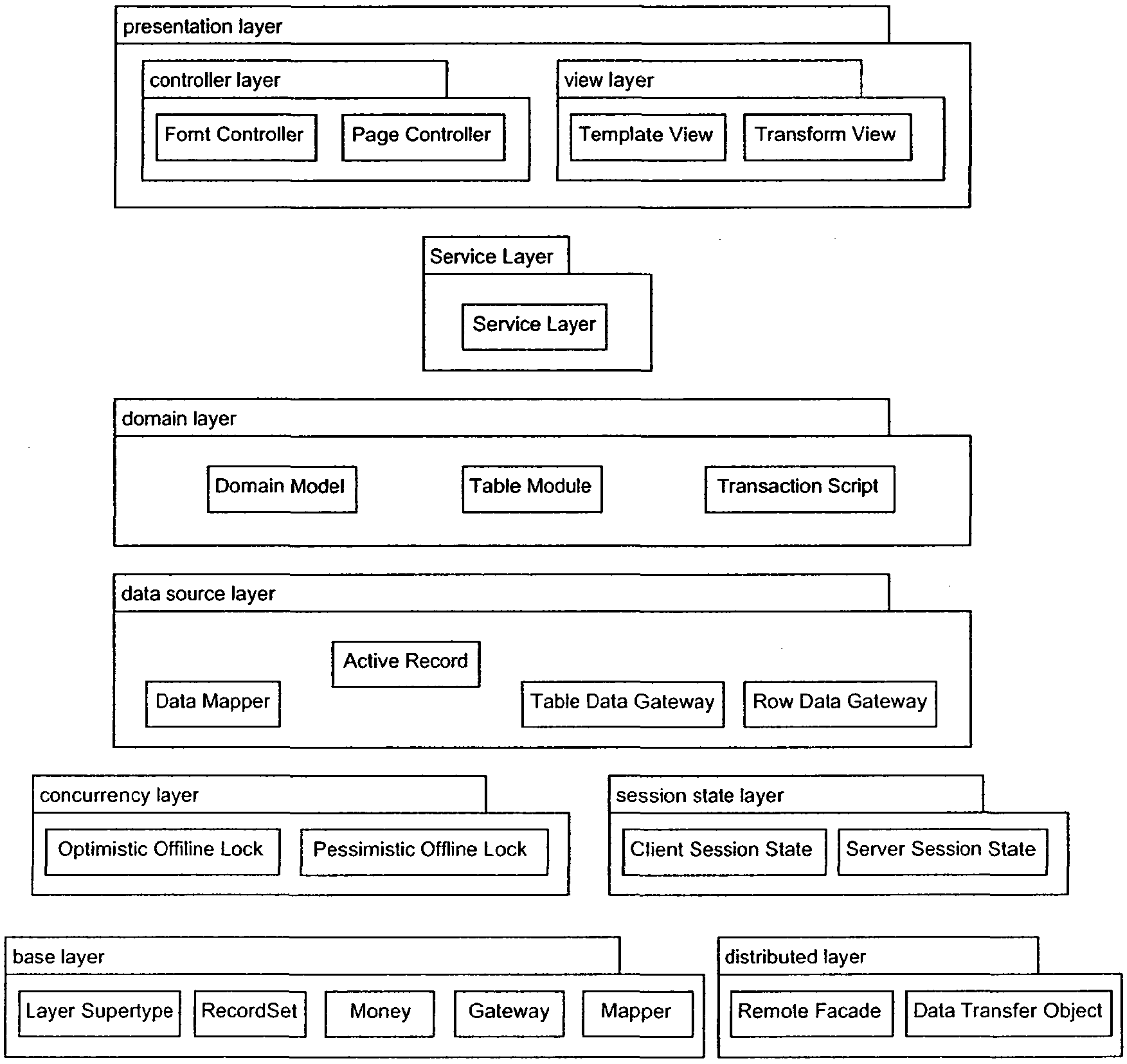


Figure 59: Selected Patterns from *PofEAA* in a Layered Architecture

A.2 Advices from the *PofEAA* Book

We have extracted 73 advices from the *PofEAA* book, shown in Table 18 to Table 21. Note that selecting the advice number (A#) and the advice classification (Type) is our choice, but the descriptions are from the book.

Table 18: Advices from the *PofEAA* Book

A#	**Type**	**Description (*PofEAA* book page#)**
A01	Syntactic	"Many of the patterns in this book are alternatives; such Page Controller and Front Controller." (p. 12)
A02	Syntactic	"A Transaction Script offers several advantages: It works well with a simple data source layer using Row Data Gateway or Table Data Gateway." (p. 25)
A03	Syntactic	"A Table Module is designed to work with a Record Set." (p. 28)
A04	Semantic	"If you have an environment like .NET or Visual Studio, then that makes a Table Module much more attractive." (p. 30)
A05	Semantic	"I don't see a reason to use Transaction Scripts in a .NET environment." (p. 30)
A06	Semantic	"However, if there's no special tooling for Record Sets, I wouldn't bother with Table Module." (p. 30)
A07	Syntactic	"These three patterns are not mutually exclusive choices. Indeed, it's quite common to use Transaction Script for some of the domain logic and Table Module or Domain Model for the rest." (p. 30)
A08	Syntactic	"A common approach in handling domain logic is to split the domain layer in two. A Service Layer is placed over an underlying Domain Model or Table Module [...] The presentation logic interacts with the domain purely through the Service Layer, which acts as an API for the application." (p. 30)
A09	Syntactic	"A domain layer that uses only Transaction Script isn't complex enough to warrant a separate [Service] layer." (p. 31)
A10	Syntactic	"The fact that Table Data Gateway fits very nicely with Record Set makes it the obvious choice if you are using Table Module."(p. 35)
A11	Syntactic	"Certainly you can use a Row Data Gateway or a Table Data Gateway with a Domain Model. For my taste, however, that can be either too much indirection or not enough." (p. 35)
A12	Syntactic	"I don't recommend using a Gateway [Row Data Gateway or Table Data Gateway] as the primary persistence mechanism for a Domain Model. If the domain logic is simple and you have a close correspondence between classes and tables, Active Record is the simple way to go. If you have something more complicated, Data Mapper is what you need." (p. 36)
A13	Syntactic	"A simple Domain Model can use Active Record, whereas a rich Domain Model requires Data Mapper." (p. 117)
A14	Syntactic	"A rich Domain Model is better for more complex logic, but is harder to map to the database." (p. 117)
A15	Syntactic	"If you have complicated and ever changing business rules involving validation, calculations, and derivations, chances are that you'll want an object model to handle them. On the other hand, if you have simple not-null checks and a couple of sums to calculate, a Transaction Script is a better bet." (p. 119)
A16	Syntactic	"If you're using Domain Model, my first choice for database interaction is Data Mapper" (p. 119)
A17	Syntactic	"Essentially you have to trade off Domain Model's ability to handle complex logic against Table Module's easier integration with the underlying table-oriented data structures[...] If the objects in a Domain Model and the database tables are relatively similar, it may be better to use a Domain Model that uses Active Record. Table Module works better than a combination of Domain Model and Active Record when other parts of the application are based on a common table-oriented data structure." (p. 128)

Table 19: Advices from the *PofEAA* Book (Cont'd)

A#	Type	Description (*PofEAA* book page#)
A18	Structural	"A Table Data Gateway has a simple interface, usually consisting of several find methods to get data from the database and update, insert, and delete methods [...] The Table Data Gateway is usually stateless." (p. 144)
A19	Syntactic	"The trickiest thing about a Table Data Gateway is how it returns information from a query [...] you can return the Record Set that comes from the SQL query." (p. 144)
A20	Syntactic	"you'll usually only see Class Table Inheritance if there's a Domain Model in your design." (p. 10)
A21	Semantic	"With a Domain Model we build a model of our domain which, at least on a first approximation, is organized primarily around the nouns in the domain." (p. 26)
A22	Syntactic	"With a Domain Model we build a model of our domain which, at least on a first approximation, is organized primarily around the nouns in the domain." (p. 26)
A23	Syntactic/ Semantic	"[for presentation layer] Your tooling may well make your choice for you. If you use Visual Studio, the easiest way to go is Page Controller and Template View. If you use Java, you have a choice of Web frameworks to consider. Popular at the moment is Struts, which will lead you to a Front Controller and a Template View." (p. 99)
A24	Semantic	"Not all systems need an Application Controller [...] A good test is this: If the machine is in control of the screen flow, you need an Application Controller; if the user is in control, you don't." (p. 58)
A25	Structural	"A Front Controller handles all calls for a Web site, and is usually structured in two parts: a Web handler and a command hierarchy. The Web handler is the object that actually receives post or get requests from the Web server." (p. 344) "The Web handler is almost always implemented as a class rather than as a server page [...] The commands are also classes rather than server pages." (p. 345)
A26	Semantic	"A related question to consider is using a single Data Transfer Object for a whole interaction versus different ones for each request [...] I might use one Data Transfer Object for most of the interaction and use different ones for a couple of requests and responses." (p. 402)
A27	Structural	"It needs to be serializable to go across the connection. Usually an assembler is used on the server side to transfer data between the DTO and any domain objects [...] Other than simple getters and setters, the Data Transfer Object is also usually responsible for serializing itself into some format that will go over the wire." (p. 401, 403)
A28	Semantic	"As optimistic locking is much easier to implement and not prone to the same defects and runtime errors as a Pessimistic Offline Lock, consider using it as the default approach to business transaction conflict management in any system you build." (p. 420)
A29	Syntactic/ Semantic	"The essence of the choice between optimistic and pessimistic locks is the frequency and severity of conflicts." (p. 68) "Whereas Pessimistic Offline Lock assumes that the chance of session conflict is high and therefore limits the system's concurrency, Optimistic Offline Lock assumes that the chance of conflict is low." (p. 417)
A30	Semantic	"The most common implementation [for Optimistic Offline Lock] is to associate a version number with each record in your system" (p. 421)
A31	Structural	"The data structure of the Active Record should exactly match that of the database: one field in the class for each column in the tabl. [...] The Active Record class typically has methods that do the following: * Construct an instance of the Active Record from a SQL result set row * Construct a new instance for later insertion into the table * Static finder methods to wrap commonly used SQL queries and return Active Record objects * Update the database and insert into it the data in the Active Record * Get and set the fields * Implement some pieces of business logic." (p. 160)
A32	Structural	"A Row Data Gateway acts as an object that exactly mimics a single record, such as one database row. In it each column in the database becomes one field [...] A Row Data Gateway should contain only database access logic and no domain logic [...] With a Row Data Gateway you're faced with the questions of where to put the find operations that generate this pattern." (p. 152)
A33	Structural	"A Record Set is usually something that you won't build yourself, provided by the vendor of the software platform you're working with. Examples include the data set of ADO.NET and the row set of JDBC 2.0 [...] Although platforms often give you a Record Set, you can create one yourself." (p. 508)

Table 20: Advices from the *PofEAA* Book (Cont'd)

A#	Type	Description (*PofEAA* book page#)
A34	Structural	"Page Controller has one input controller for each logical page of the Web site. [...] The basic idea behind a Page Controller is to have one module on the Web server act as the controller for each page on the Web site." (p. 333)
A35	Syntactic	"It's not uncommon to have a site where some requests are dealt with by Page Controllers and others are dealt with by Front Controllers." (p. 335)
A36	Syntactic	"Add remotability when you need it (if ever) by putting Remote Facades on your Service Layer." (p. 135)
A37	Structural	"A Transaction Script organizes all this logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each transaction will have its own Transaction Script, although common subtasks can be broken into subprocedures." (p. 113)
A38	Syntactic	"Usually you use Table Module with a backing data structure that's table oriented. The tabular data is normally the result of a SQL call and is held in a Record Set that mimics a SQL table. [...] The Table Module may include queries as factory methods. The alternative is a Table Data Gateway." (p. 126, 127)
A39	Structural	"A Table Module organizes domain logic with one class per table in the database [...] Each Table Module class has a data member of a data table." (p. 125)
A40	Structural	"A type that acts as the supertype for all types in its layer [...] All you need is a superclass for all the objects in a layer [...] Use Layer Supertype when you have common features from all objects in a layer." (p. 475)
A41	Structural	"The Data Mapper [...] separates the in-memory objects from the database [...] The separation between domain and data source is the main function of a Data Mapper [...] A simple Data Mapper would just map a database table to an equivalent in-memory class on a field-to-field basis [...] [for] inserts and updates, the database mapping layer needs to understand what objects have changed, which new ones have been created, and which ones have been destroyed [...] We'll use the simple case here, where the Person Mapper class also implements the finder and Identity Map." (p. 165)
A42	Structural	"An object model of the domain that incorporates both behavior and data. [...] A Domain Model mingles data and process, has multivalued attributes and a complex web of associations, and uses inheritance." (p. 116)
A43	Structural	"The basic idea is to have a Money class with fields for the numeric amount and the currency [...] Money needs arithmetic operations so that you can use money objects as easily as you use numbers." (p. 488)
A44	Structural	"Remote Facade contains no domain logic [...] In a simple case, like an address object, a Remote Facade replaces all the getting and setting methods of the regular address object with one getter and one setter, often referred to as bulk accessors." (p. 389)
A45	Structural	"The two basic implementation variations [for Service Layer] are the domain facade approach and the operation script approach. In the domain facade approach a Service Layer is implemented as a set of thin facades over a Domain Model [...] The thin facades establish a boundary and set of operations through which client layers interact with the application, exhibiting the defining characteristics of Service Layer." (p. 134)
A46	Semantic	"The parameter list of the insert method must be a subset of the parameter list of the update method." (p. 144)
A47	Syntactic/ Semantic	"The easier question to answer is probably when not to use it. You probably don't need a Service Layer if your application's business logic will only have one kind of client-say, a user interface-and its use case responses don't involve multiple transactional resources." (p. 137)
A48	Syntactic/ Semantic	"Hence, we get to my First Law of Distributed Object Design: Don't distribute your objects! " (p. 89)
A49	Syntactic	"For this book I'm centering my discussion around an architecture of three primary layers: presentation, domain, and data source." (p. 19)
A50	Syntactic	"There are two patterns for the input controller. The most common is an input controller object for every page on your Web site. In the simplest case this Page Controller can be a server page itself, combining the roles of view and input controller [...] A server page can handle the request, delegating a separate helper object to decide what to do with it. Front Controller (344) goes further in this separation by having only one object handling all requests." (p. 61)

Table 21: Advices from the *PofEAA* Book (Cont'd)

A#	Type	Description (*PofEAA* book page#)
A51	Syntactic	"Often you'll find that there isn't quite a one-to-one relationship between Page Controllers and views." (p. 61)
A52	Semantic	"On the view front the choice between Template View and Transform View depends on whether your team uses server pages [HTML] or XSLT in programming. You can write a Transform View in any language; at the moment, however, the dominant choice is XSLT." (p. 99, 361)
A53	Syntactic	"Since it's a form of Mapper, Data Mapper itself is even unknown to the domain layer." (p. 165)
A54	Syntactic	"[Table Data Gateway is] An object that acts as a Gateway to a database table. [...] I see this pattern [Table Data Gateway] as a particular usage of the more general Gateway concept." (p. 144, 146)
A55	Syntactic	"[The problem here is how to synchronize it with other modules. is] An object that acts as a Gateway to a single record in a data source." (p. 152)
A56	Syntactic	"The most common case of a mapping layer that we run into is in a Data Mapper [...] Thus, in enterprise applications we mostly find Mapper used for interactions with a database, as in Data Mapper." (p. 473, 474)
A57	Structural	"The best way to work is to compose the dynamic Web page as you do a static page but put in markers that can be resolved into calls to gather dynamic information. Since the static part of the page acts as a template for the particular response, I call this a Template View [...] The key to avoiding scriptlets is to provide a regular object as a helper to each page." (p. 350, 352)
A58	Syntactic	"For implementing the view in Model View Controller the main choice is between Template View and Transform View." (p. 354)
A59	Structural	"A Transform View is organized around separate transforms for each kind of input element." (p. 361)
A60	Semantic	"A Transaction Script offers several advantages: It's a simple procedural model that most developers understand." (p. 25)
A61	Syntactic	"My preference is thus to have the thinnest Service Layer you can, if you even need one." (p. 32)
A62	Syntactic/ Semantic	"So everything should be stateless, right? Well, it would be if it could be." (p. 82)
A63	Syntactic/ Semantic	"Concurrency is one of the most tricky aspects of software development. Whenever you have multiple processes or threads manipulating the same data, you run into *concurrency problems*." (p. 63)
A64	Syntactic	"In organizing domain logic I've separated it into three primary patterns: Transaction Script, Domain Model, and Table Module." (p. 25)
A65	Syntactic	"On the view side there are three patterns to think about: Transform View, Template View, and Two Step View." (p. 58)
A66	Syntactic	"In broad terms there are two forms of concurrency control that we can use: optimistic and pessimistic." (p. 67)
A67	Syntactic	"So, how do you store session state once you know you have to have it? I divide the options into three blurred but basic choices. Client Session State [...] Database Session State [...] Database Session State." (p. 84)
A68	Syntactic	"Hand in hand with Remote Facade is Data Transfer Object." (p. 92)
A69	Syntactic	"While most of these patterns are truly for enterprise applications, those in the base patterns chapter (Chapter 18) are more general and localized." (p. 11)
A70	Structural	"Wrap all the special API code into a class whose interface looks like a regular object. Other objects access the resource through this Gateway, which translates the simple method calls into the appropriate specialized API." (p. 466)
A71	Structural	"the objects that a Mapper separates aren't even aware of the mapper." (p. 474)
A72	Structural	"You almost always have to use Client Session State for session identification [Session ID]." (p. 457)
A73	Structural	"In the simplest form of this pattern a session object is held in memory on an application server." (p. 458)
A74	Syntactic	"If you use procedural scripts as your view, you can write the code in the style of either Transform View or Template View or in some interesting mix of the two." (p. 59)

A.3 *PofEAA* Rule Set

The advices shown in the previous section, have been interpreted into formal rules, *Structural, Syntactic*, and *Semantic*, using the formalisms defined in Chapter 3. The obtained formal rules are shown in the following sections, respectively.

Note that for the Syntactic and Semantic rules, we have used conditions that are specified in the following. Also, the numbers given at the end of each rule (:Axx) shows the advice(s) from the Section A.2 that is referenced to define that rule.

- C11: Tool is .Net
- C12: Tool is Java
- C21: Domain structure is Simple
- C22: Domain structure is Moderate
- C23: Domain structure is Complex
- C31: Designer is Novice
- C32: Designer is Intermediate
- C33: Designer is Expert
- C41: Designer wants Service Layer
- C42: Designer wants Distributed Layer
- C43: Designer wants Concurrency Layer
- C44: Designer wants Session State Layer
- C51: Chance of conflict is Low
- C52: Chance of conflict is High
- C61: View is built using HTML
- C62: View is built using XSLT

A.3.1 Part I: Structural Rules

The structural rules for 23 selected patterns from *PofEAA* are shown in the following 23 figures (Figure 60 to Figure 82). Note that, "The Intent and the Sketch," if present, are from the *PofEAA* book. The CRUD is an abbreviation for DB operations Create, Read, Update, and Delete, but in the following, by CRUD, we mean `find, insert, delete`, and `update`.

Front Controller

1. There is a `Front Controller` (=`Handler`) class in the model.
2. There are at least two operations (`doGet` and `doPost`) in the Handler class.
3. The `Handler` class has a client dependency to a `Command` class.
4. The `Command` class is abstract.
5. The `Command` class has at least one `process` operation.
6. The `Command` class has at least one `Concrete Command` child class.
7. A `Concrete Command` class is concrete.
8. A `Concrete Command` class has at least one `process` operation.

A contoller that handles all the requests for a Web site.

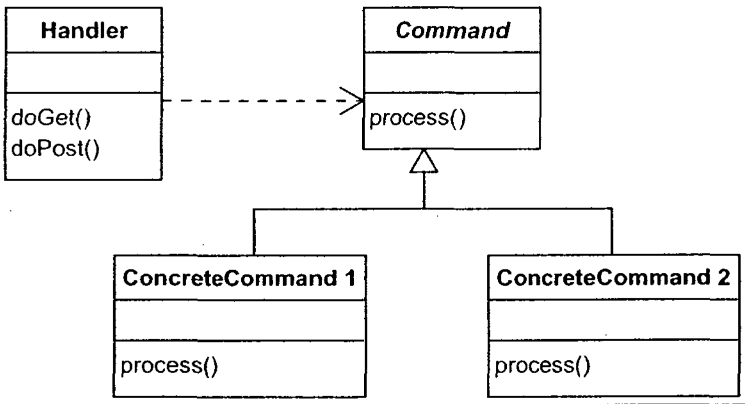


Figure 60: Structural Rules, Intent, and Sketch of Front Controller Pattern

Page Controller

1. There is a `Page Controller` class in the model.
2. There are at least two operations (`doGet` and `doPost`) in the `Page Controller` class.

An object that handles a request for a specific page or action on a Web site.

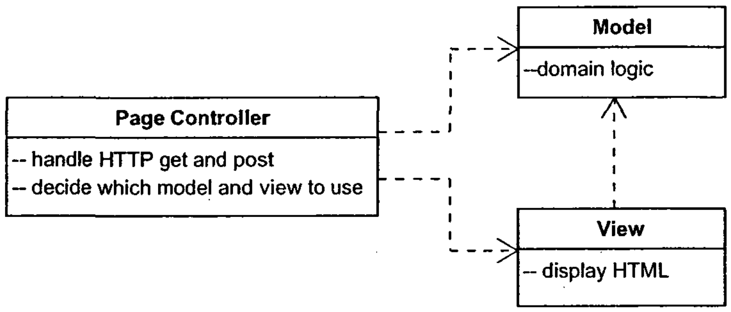
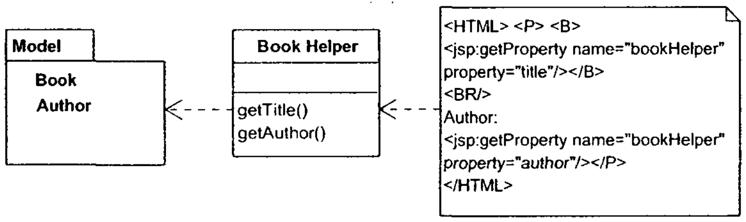

Model
--domain logic
Page Controller
-- handle HTTP get and post
-- decide which model and view to use
View
-- display HTML

Figure 61: Structural Rules, Intent, and Sketch of Page Controller Pattern

Template View

1. There is a `Template View` class in the model.
2. The `Template View` class has a client dependency to a `Helper` class.

Renders information into HTML by embedding markers in an HTML page.



```
<HTML> <P> <B>
<jsp:getProperty name="bookHelper"
property="title"/></B>
<BR/>
Author:
<jsp:getProperty name="bookHelper"
property="author"/></P>
</HTML>
```

Figure 62: Structural Rules, Intent, and Sketch of Template View Pattern

Transform View

1. There is a `Transform View` class in the model.
2. There is at least one `transform` operation in the `Transform View` class.

A view that processes domain data element by element and transforms it into HTML.

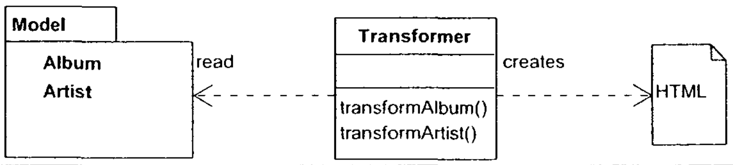


Figure 63: Structural Rules, Intent, and Sketch of Transform View Pattern

Service Layer

(Note: The Sketch given here is an adapted form of the Sketch given in [Fow02, p. 133])

1. There is a `Service Layer` class in the model.
2. There is at least one operation in the `Service Layer` class.
3. All the operations in the `Service Layer` class must be public.

Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

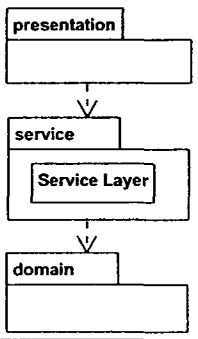


Figure 64: Structural Rules, Intent, and Sketch of Service Layer Pattern

Domain Model

1. There is a `Domain Model` class in the model.
2. There is at least one operation in the `Domain Model` class.
3. There is at least one attribute in the `Domain Model` class.

An object model of the domain that incorporates both behavior and data.

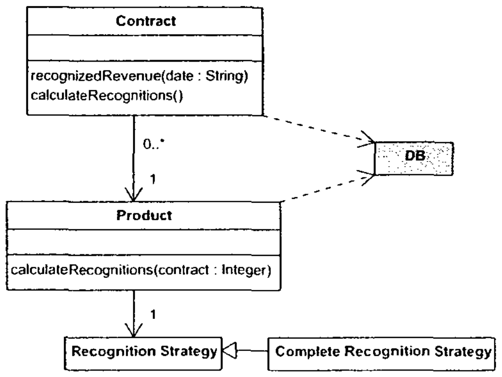

Contract
recognizedRevenue(date : String)
calculateRecognitions()
0..*
1
DB
Product
calculateRecognitions(contract : Integer)
1
Recognition Strategy
Complete Recognition Strategy

Figure 65: Structural Rules, Intent, and Sketch of Domain Model Pattern

Table Module

1. There is a `Table Module` class in the model.
2. There is at least one operation in the `Table Module` class.
3. There is at least one `data table` attribute in the `Table Module` class.

A single instance that handles the business logic for all rows in a database table or view.

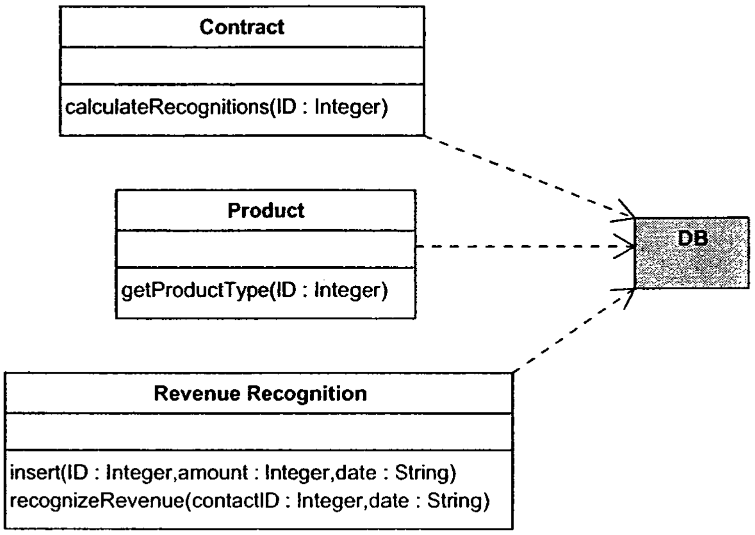

Contract
calculateRecognitions(ID : Integer)
Product
getProductType(ID : Integer)
DB
Revenue Recognition
insert(ID : Integer,amount : Integer,date : String)
recognizeRevenue(contactID : Integer,date : String)

Figure 66: Structural Rules, Intent, and Sketch of Table Module Pattern

Transaction Script

1. There is a `Transaction Script` class in the model.
2. There is at least one operation in the `Transaction Script` class.

Organizes business logic by procedures where each procedure handles a single request from the presentation.

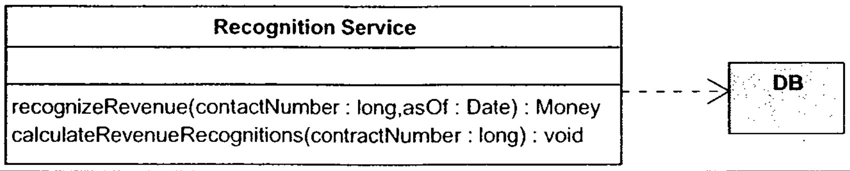

Recognition Service
recognizeRevenue(contactNumber : long,asOf : Date) : Money
calculateRevenueRecognitions(contractNumber : long) : void
DB

Figure 67: Structural Rules, Intent, and Sketch of Transaction Script Pattern

Data Mapper

1. There is a `Data Mapper` class in the model.
2. The `Data Mapper` class is stateless, e.g., has no public attribute.
3. There are CRUD operations in the `Data Mapper` class.
4. The `Data Mapper` class has a client dependency to at least one other class.

A layer of Mappers (473) that moves data between objects and a database while keeping them independent of each other and the mapper itself.

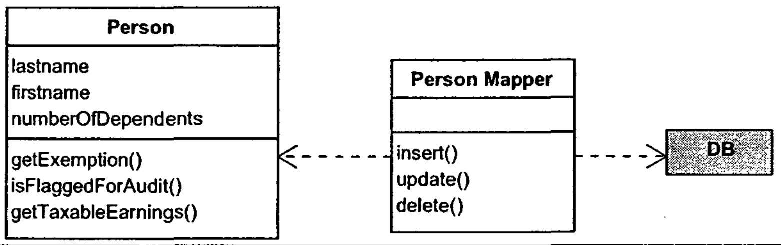


Figure 68: Structural Rules, Intent, and Sketch of Data Mapper Pattern

Active Record

1. There is an `Active Record` class in the model.
2. There is at least one attribute in the `Active Record` class.
3. There are CRUD operations in the `Active Record` class.

An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

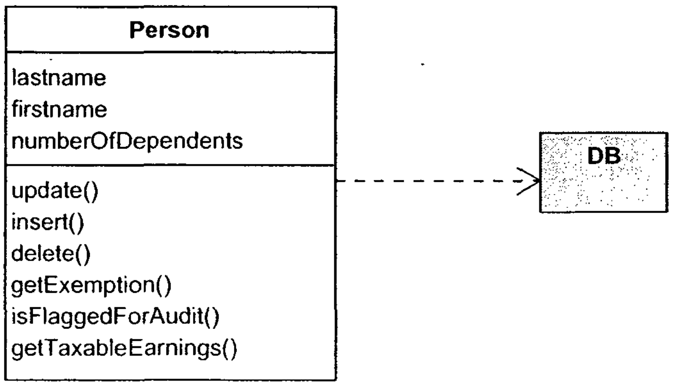


Figure 69: Structural Rules, Intent, and Sketch of Active Record Pattern

Table Data Gateway

1. There is a `Table Data Gateway` class in the model.
2. There are CRUD operations in the `Table Data Gateway` class.

An object that acts as a Gateway (466) to a database table.
One instance handles all the rows in the table.

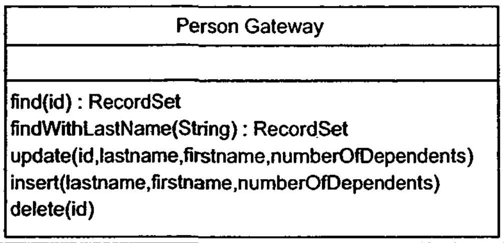

Person Gateway
find(id) : RecordSet
findWithLastName(String) : RecordSet
update(id,lastname,firstname,numberOfDependents)
insert(lastname,firstname,numberOfDependents)
delete(id)

Figure 70: Structural Rules, Intent, and Sketch of Table Data Gateway Pattern

Row Data Gateway

1. There is a `Row Data Gateway` class in the model.
2. There are at least three operations: `insert, delete` and `update` in the `Row Data Gateway` class.
3. There is a `Finder` class as a client for the `Row Data Gateway` class.
4. There is at least one `find` operation in the `Finder` class.

An object that acts as a Gateway (466) to a single record in a data source.
There is one instance per row.

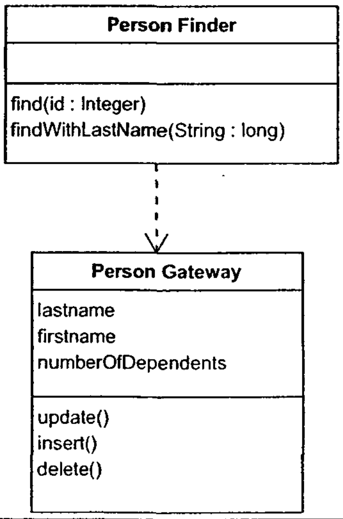


Figure 71: Structural Rules, Intent, and Sketch of Row Data Gateway Pattern

Remote Facade

1. There is a `Remote Facade` class in the model.
2. The `Remote Facade` class is a client of a supplier class.
3. There are at least two `getter` and two `setter` operations in the supplier class.
4. There are at least two bulk accessor operations (`getBulk` and `setBulk`) in the `Remote Facade` class.

Provides a coarse-grained facade on fine-grained objects to improve efficiency over a network.

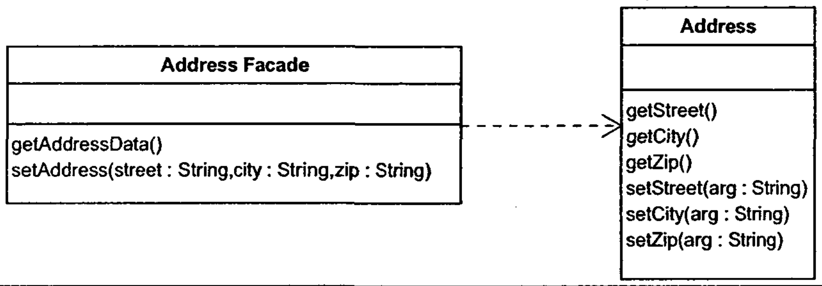


Figure 72: Structural Rules, Intent, and Sketch of Remote Facade Pattern

Data Transfer Object

1. There is a `Data Transfer Object` class in the model.
2. There is at least one `getter` and one `setter` operation in the `Data Transfer Object` class.
3. There is one `serialize` and one `deserialize` operation in the `Data Transfer Object` class.
4. There is an `Assembler` class a client for the `Data Transfer Object` class.

Provides a coarse-grained facade on fine-grained objects to improve efficiency over a network.

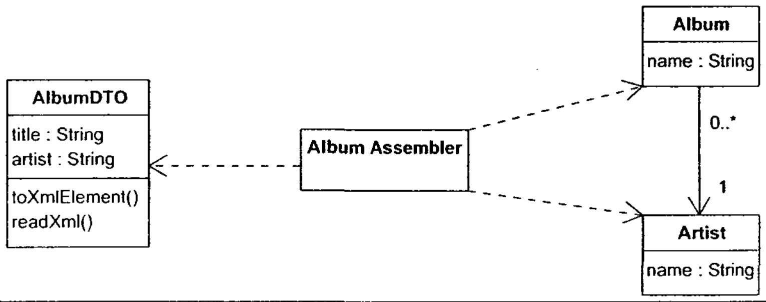


Figure 73: Structural Rules, Intent, and Sketch of Data Transfer Object Pattern

Optimistic Offline Lock

1. There is a `Optimistic Offline Lock` class in the model.
2. There is at least one `version` attribute in the `Optimistic Offline Lock` class.

Figure 74: Structural Rules of Optimistic Offline Lock Pattern

Pessimistic Offline Lock

1. There is a `Pessimistic Offline Lock` class in the model.
2. There is at least one `lock` operation in the `Pessimistic Offline Lock` class.

Figure 75: Structural Rules, Intent, and Sketch of Pessimistic Offline Lock Pattern

Client Session State

1. There is a `Client Session State` class in the model.
2. There is at least one `Session ID` attribute in the `Client Session State` class.

Figure 76: Structural Rules of Client Session State Pattern

Server Session State

1. There is a `Server Session State` class in the model.
2. There is at least one `Session ID` attribute in the `Server Session State` class.
3. There are at least two operations (`serialize` and `deserialize`) in the `Server Session State` class.

Figure 77: Structural Rules of Server Session State Pattern

Layer Supertype

1. There is a `Layer Supertype` class in the model.
2. There is at least one operation in the `Layer Supertype` class.
3. There is at least one child for the `Layer Supertype` class.
4. All the children of the `Layer Supertype` class must be of the same type.

Figure 78: Structural Rules of Layer Supertype Pattern

Record Set

1. There is a `Record Set` class in the model.
2. The `Record Set` class must have a navigable one-to-many composite association towards a `Table` class.
3. The `Table` class has a navigable one-to-many composite association towards a `Row` class.
4. The `Table` class has a navigable one-to-many composite association towards a `Column` class

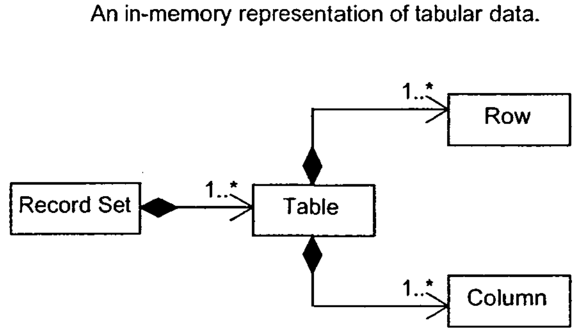


Figure 79: Structural Rules, Intent, and Sketch of Record Set Pattern

Money

1. There is a `Money` class in the model.
2. There are two attributes `amount` and `currency` in the `Money` class.
3. There is at least one operation in the `Money` class.

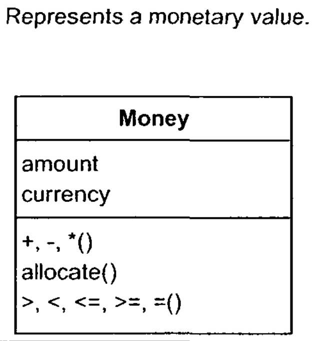


Figure 80: Structural Rules, Intent, and Sketch of Money Pattern

Gateway

1. There is a `Gateway` class in the model.
2. There is at least one operation in the `Gateway` class.
3. There is at least one client class for the `Gateway` class.
4. There is at least one supplier class for the `Gateway` class.

An object that encapsulates access to an external system or resource.

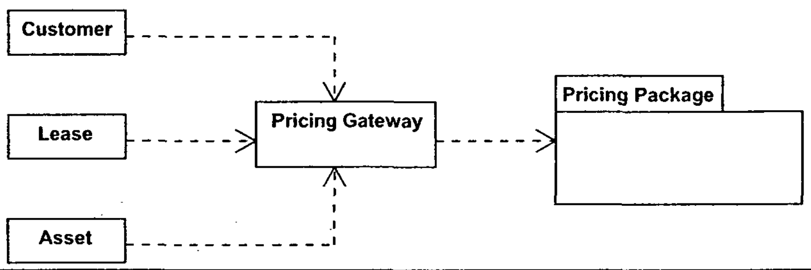


Figure 81: Structural Rules, Intent, and Sketch of Gateway Pattern

Mapper

1. There is a `Mapper` class in the model.
2. There is at least one operation in the `Mapper` class.
3. There is at least one supplier class for the `Mapper` class.
4. There is no client class for the `Mapper` class.

An object that sets sup a communication between two independent objects.

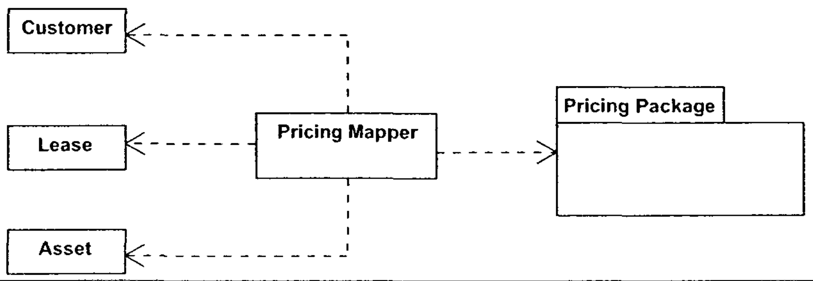


Figure 82: Structural Rules. Intent, and Sketch of Mapper Pattern

A.3.2 Part II: Syntactic Rules (Pattern Organizations)

This section uses the formalism introduced in Section 3.3.2 to define syntactic rules that show the layers of the system and the placement of the patterns inside the layers. The references given at the end of each rule, in the form of ":Axx," refers to the advices in Section A.2.

pofeaa model $\supset$ *main layer* . *auxiliary layer** :A49

main layer $\supset$ *presentation* . *service*$^{?(C41)}$. *domain* . *datasource* :A08, A47, A49

presentation $\supset$ *controller* . *view* :A22

auxiliary layer $\supset$ *base** , *distributed*$^{?(C42)}$, *concurrency*$^{?(C43)}$, *sessionstate*$^{?(C44)}$:A48, A63, A62

controller $\ni$ *Page Controller* , *Front Controller* :A50

view $\ni$ *Template View* , *Transform View* :A65

service $\ni$ *Service Layer* :A61

domain $\ni$ *Domain Model*, *Table Module*, *Transaction Script* :A64

datasource $\ni$ *Data Mapper*, *Active Record*, *Table Data Gateway*, *Row Data Gateway* :A12

base $\ni$ *Record Set*, *Layer Supertype*, *Money*, *Mapper*, *Gateway* :A11

distributed $\ni$ *Remote Facade*, *Data Transfer Object* :A68

concurrency $\ni$ *Optimistic Offline Lock*, *Pessimistic Offline Lock* :A66

sessionstate $\ni$ *Client Session State*, *Server Session State* :A67

A.3.3 Part III: Syntactic Rules (Pattern Relationships)

This section uses the formalism introduced in Section 3.3.2 to define the relationship between patterns. The references given at the end of each rule, in the form of ":Axx," refers to the advices in Section A.2.

$\overline{\textit{Page Controller}} \rightarrow$ *Template View* | *Transform View* :A35, A51

$\overline{\textit{Front Controller}} \rightarrow$ *Template View* | *Transform View* :A35

$\overline{\textit{Page Controller}} \overset{C11}{\rightarrow}$ *Template View* :A23

$\overline{\textit{Front Controller}} \overset{C12}{\rightarrow}$ *Template View* :A23

Template View $\overset{C41}{\rightarrow}$ *Service Layer*

Transform View $\overset{C41}{\rightarrow}$ *Service Layer*

Service Layer $\rightarrow$ *Domain Model* | *Table Module* :A08, A09

Template View $\overset{\neg C41}{\rightarrow}$ *Domain Model* | *Table Module* | *Transaction Script* :A07

Transform View $\overset{\neg C41}{\rightarrow}$ *Domain Model* | *Table Module* | *Transaction Script* :A07

Page Controller $\overset{\neg C41}{\rightarrow}$ *Domain Model* | *Table Module* | *Transaction Script* ??? :A07

Front Controller $\overset{\neg C41}{\rightarrow}$ *Domain Model* | *Table Module* | *Transaction Script* ??? :A07

Domain Model $\overset{C21}{\rightarrow}$ *Active Record* :A12, A13

Domain Model $\overset{C23}{\rightarrow}$ *Data Mapper* :A12, A13

Table Module $\rightarrow$ *Table Data Gateway* | *Row Data Gateway* :A38

Table Module $\rightarrow$ *Table Data Gateway* {C113} :A10

Transaction Script $\rightarrow$ *Table Data Gateway* | *Row Data Gateway* :A2

Table Data Gateway $\rightarrow$ *Record Set* {C111} :A19

Table Data Gateway $\overset{C42}{\rightarrow}$ *Data Transfer Object* :A19

Data Mapper $\overset{datasource}{\leftrightarrow}$ *Active Record* :A13

Table Data Gateway $\overset{datasource}{\leftrightarrow}$ *Row Data Gateway* {C112}

Optimistic Offline Lock $\overset{concurrency}{\leftrightarrow}$ *Pessimistic Offline Lock* {C112}

Client Session State $\overset{sessionstate}{\leftrightarrow}$ *Server Session State* {C112}

FrontController $\upharpoonright$ *Controller* :A50

PageController $\upharpoonright$ *Controller* :A50

Data Mapper $\upharpoonright$ *Mapper* :A53

Table Data Gateway $\upharpoonright$ *Gateway* :A54

Row Data Gateway $\upharpoonright$ *Gateway* :A55

C111: Return type of every find() operation in the Table Data Gateway pattern is Record Set

C112: Two patterns are applied for the same unit of work

C113: There is special tooling for Record Sets

A.3.4 Part IV: Semantic Rules

This section uses the formalism introduced in Section 3.3.3 to define the semantic rules governing the application of patterns. The references given at the end of each rule, in the form of ":Axx," refers to the advices in Section A.2.

Page Controller $\approx$ {C11} :A23

Front Controller $\approx$ {C12} :A23

Template View $\approx$ {C61} :A52

Transform View $\approx$ {C62} :A52

Domain Model $\approx$ {C12 and and C23 C33} :A??

Transaction Script $\not\approx$ {C11} :A05

Transaction Script $\approx$ {C21 and C31} :A60

Table Module $\approx$ {C11} :A04

Table Module $\not\approx$ {C122} :A06

Table Data Gateway $\approx$ {*insert() parameter list* $\subseteq$ *update() parameter list*} :A46?

Active Record $\approx$ *Template View* {C121} :A??

Service Layer $\approx$ {C41} :A47

Remote Facade $\approx$ {C42} :A48

Data Transfer Object $\approx$ {C42} :A48

Optimistic Offline Lock $\approx$ {C43 and C51} :A29, A63

Pessimistic Offline Lock $\approx$ {C43 and C52} :A29, A63

Client Session State $\approx$ {C44} :A62

Server Session State $\approx$ {C44} :A62

C121: The parameters of the operations of the Active Record pattern must match with the attributes of Template View

C122: There is no special tooling for Record Sets

A.4 *PofEAA* UML Profile

A.4.1 Stereotypes

Stereotypes are shown by the white boxes in Figure 83. The gray boxes are the UML meta-classes. An arrows from a stereotype to a UML meta class, must be interpreted as an extension.

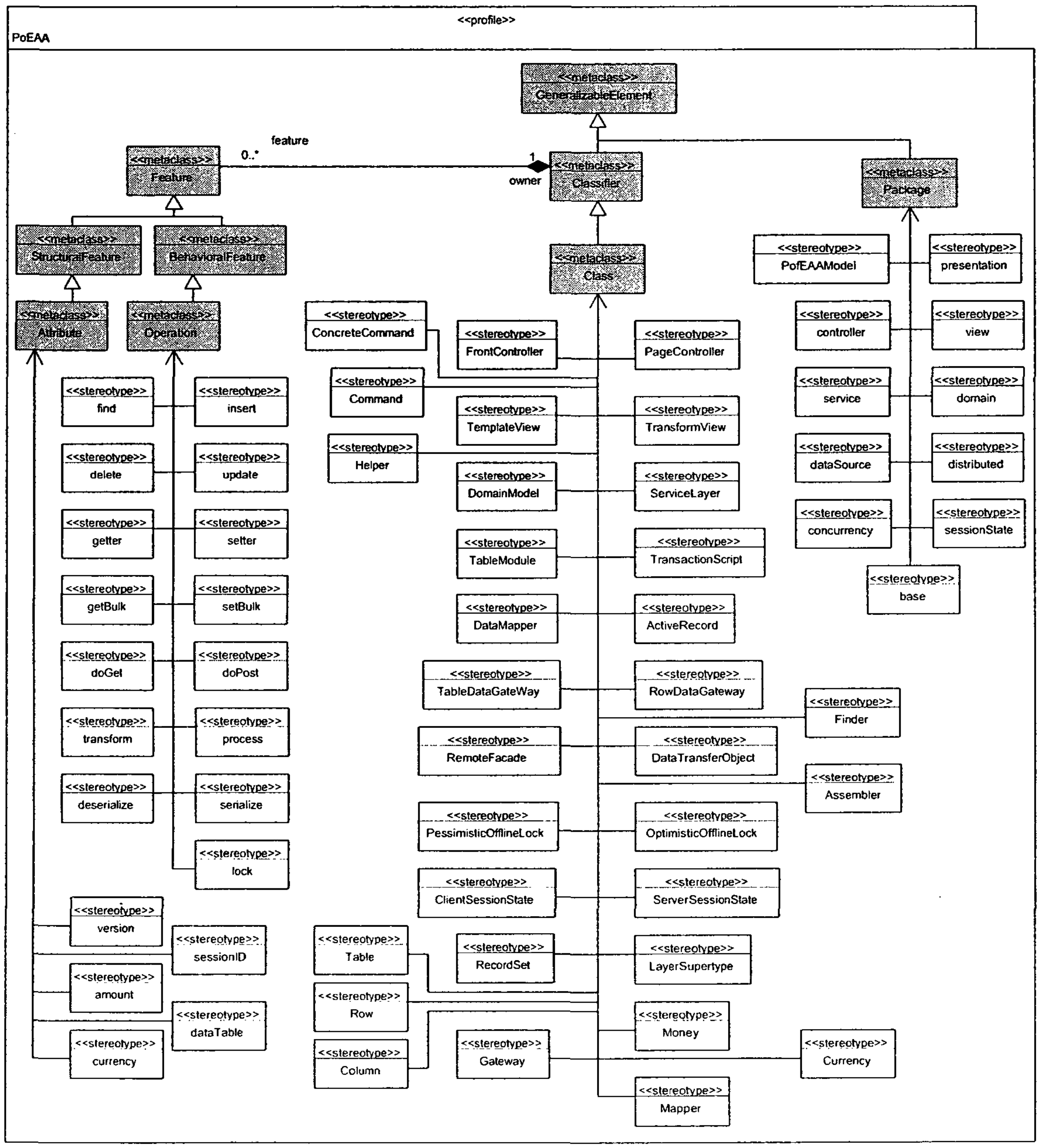


Figure 83: Mapping the *PofEAA* Meta-model into the UML Meta-model

A.4.2 Tagged Values

Tagged Values are represented in Table 22.

Table 22: Tagged Values for stereotype «PofEAAModel» in *PofEAA* UML Profile

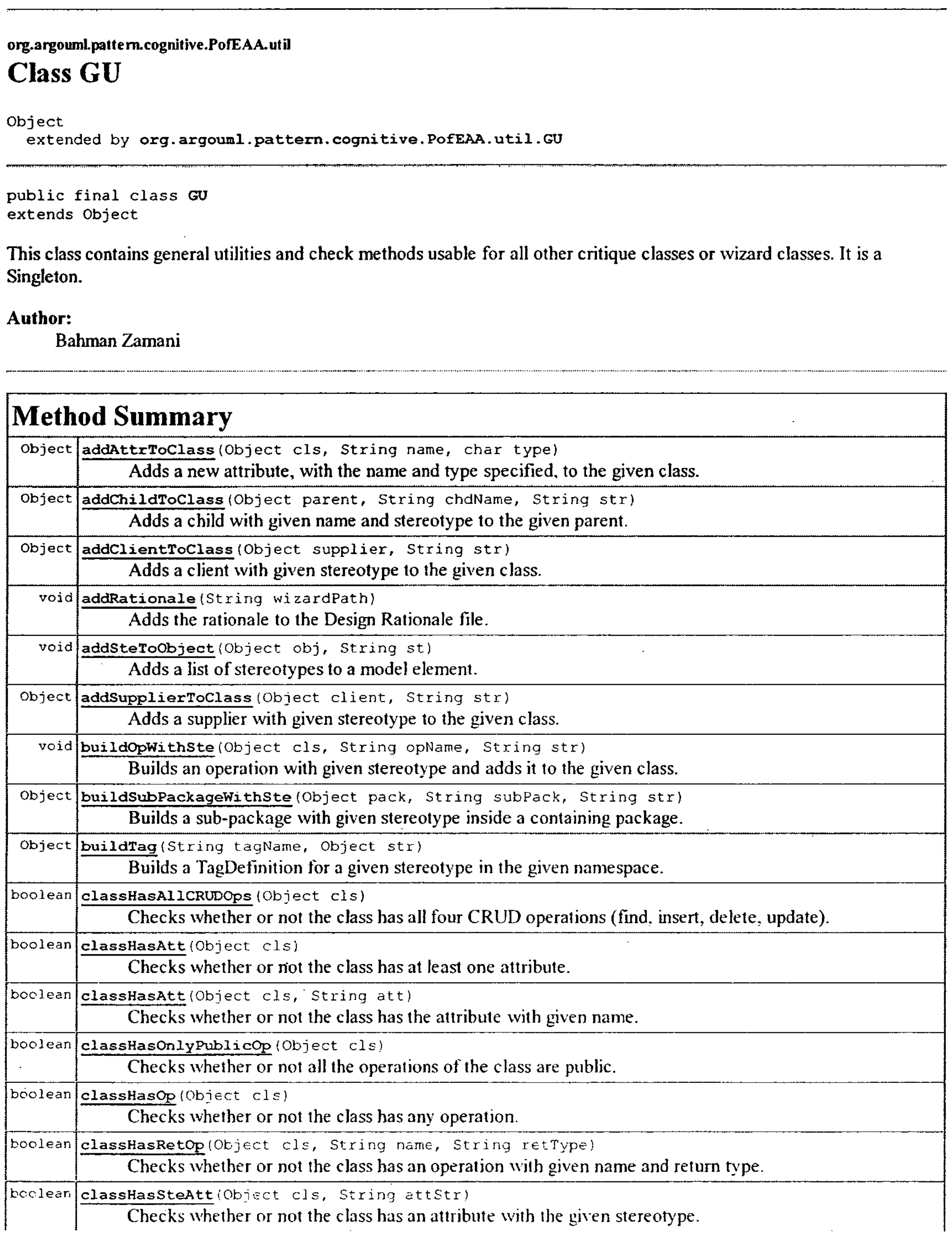
Tag	**Type**	**Mult.**	**Values**
ServiceLayer	String	[0..1]	Yes , No
DistributedLayer	String	[0..1]	Yes , No
ConcurrencyLayer	String	[0..1]	Yes , No
SessionStateLayer	String	[0..1]	Yes , No
ChanceOfConflict	String	[0..1]	Low , High
ViewBuilt	String	[0..1]	HTML , XSLT
Tool	String	[0..1]	Java , .Net
Complexity	String	[0..1]	Simple , Moderate, Complex
Expertise	String	[0..1]	Novice , Intermediate , Expert

A.4.3 Constraints

Constraints are written in Java as ArgoUML *critic* classes. The source code of some of these *critics* are shown in Section A.5.

A.5 Source Code Excerpts

A.5.1 JavaDoc for General Utility Singleton Class (GU.java)



org.argouml.pattern.cognitive.PofEAA.util

Class GU

```
Object
  extended by org.argouml.pattern.cognitive.PofEAA.util.GU
```

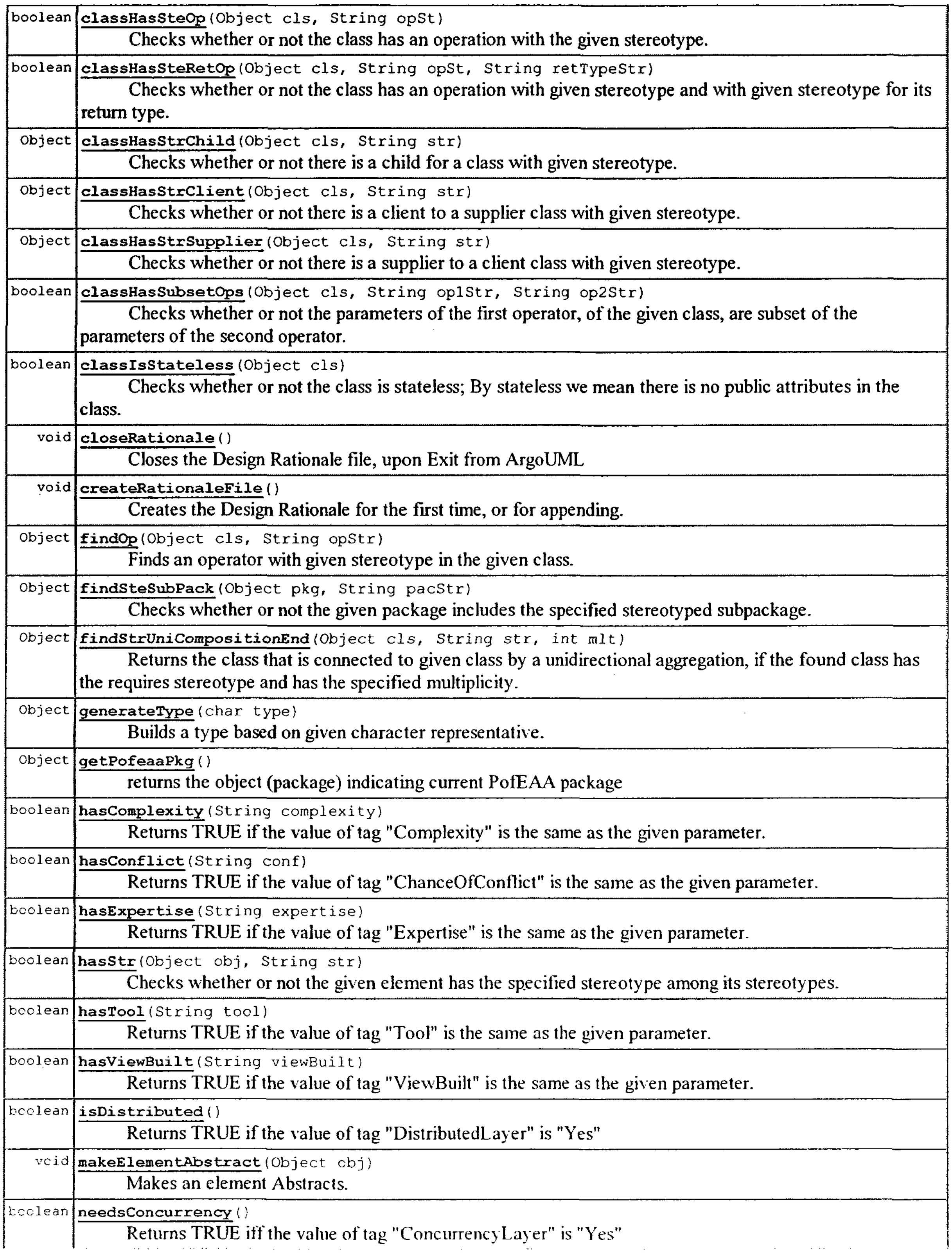
```
public final class GU
extends Object
```

This class contains general utilities and check methods usable for all other critique classes or wizard classes. It is a Singleton.

Author:
Bahman Zamani

Method Summary	
Object	**addAttrToClass**(Object cls, String name, char type) Adds a new attribute, with the name and type specified, to the given class.
Object	**addChildToClass**(Object parent, String chdName, String str) Adds a child with given name and stereotype to the given parent.
Object	**addClientToClass**(Object supplier, String str) Adds a client with given stereotype to the given class.
void	**addRationale**(String wizardPath) Adds the rationale to the Design Rationale file.
void	**addSteToObject**(Object obj, String st) Adds a list of stereotypes to a model element.
Object	**addSupplierToClass**(Object client, String str) Adds a supplier with given stereotype to the given class.
void	**buildOpWithSte**(Object cls, String opName, String str) Builds an operation with given stereotype and adds it to the given class.
Object	**buildSubPackageWithSte**(Object pack, String subPack, String str) Builds a sub-package with given stereotype inside a containing package.
Object	**buildTag**(String tagName, Object str) Builds a TagDefinition for a given stereotype in the given namespace.
boolean	**classHasAllCRUDOps**(Object cls) Checks whether or not the class has all four CRUD operations (find, insert, delete, update).
boolean	**classHasAtt**(Object cls) Checks whether or not the class has at least one attribute.
boolean	**classHasAtt**(Object cls, String att) Checks whether or not the class has the attribute with given name.
boolean	**classHasOnlyPublicOp**(Object cls) Checks whether or not all the operations of the class are public.
boolean	**classHasOp**(Object cls) Checks whether or not the class has any operation.
boolean	**classHasRetOp**(Object cls, String name, String retType) Checks whether or not the class has an operation with given name and return type.
boolean	**classHasSteAtt**(Object cls, String attStr) Checks whether or not the class has an attribute with the given stereotype.

Figure 84: GU class Javadoc, page 1



boolean	**classHasSteOp**(Object cls, String opSt) Checks whether or not the class has an operation with the given stereotype.
boolean	**classHasSteRetOp**(Object cls, String opSt, String retTypeStr) Checks whether or not the class has an operation with given stereotype and with given stereotype for its return type.
Object	**classHasStrChild**(Object cls, String str) Checks whether or not there is a child for a class with given stereotype.
Object	**classHasStrClient**(Object cls, String str) Checks whether or not there is a client to a supplier class with given stereotype.
Object	**classHasStrSupplier**(Object cls, String str) Checks whether or not there is a supplier to a client class with given stereotype.
boolean	**classHasSubsetOps**(Object cls, String op1Str, String op2Str) Checks whether or not the parameters of the first operator, of the given class, are subset of the parameters of the second operator.
boolean	**classIsStateless**(Object cls) Checks whether or not the class is stateless; By stateless we mean there is no public attributes in the class.
void	**closeRationale**() Closes the Design Rationale file, upon Exit from ArgoUML
void	**createRationaleFile**() Creates the Design Rationale for the first time, or for appending.
Object	**findOp**(Object cls, String opStr) Finds an operator with given stereotype in the given class.
Object	**findSteSubPack**(Object pkg, String pacStr) Checks whether or not the given package includes the specified stereotyped subpackage.
Object	**findStrUniCompositionEnd**(Object cls, String str, int mlt) Returns the class that is connected to given class by a unidirectional aggregation, if the found class has the requires stereotype and has the specified multiplicity.
Object	**generateType**(char type) Builds a type based on given character representative.
Object	**getPofeaaPkg**() returns the object (package) indicating current PofEAA package
boolean	**hasComplexity**(String complexity) Returns TRUE if the value of tag "Complexity" is the same as the given parameter.
boolean	**hasConflict**(String conf) Returns TRUE if the value of tag "ChanceOfConflict" is the same as the given parameter.
boolean	**hasExpertise**(String expertise) Returns TRUE if the value of tag "Expertise" is the same as the given parameter.
boolean	**hasStr**(Object obj, String str) Checks whether or not the given element has the specified stereotype among its stereotypes.
boolean	**hasTool**(String tool) Returns TRUE if the value of tag "Tool" is the same as the given parameter.
boolean	**hasViewBuilt**(String viewBuilt) Returns TRUE if the value of tag "ViewBuilt" is the same as the given parameter.
boolean	**isDistributed**() Returns TRUE if the value of tag "DistributedLayer" is "Yes"
void	**makeElementAbstract**(Object obj) Makes an element Abstracts.
boolean	**needsConcurrency**() Returns TRUE iff the value of tag "ConcurrencyLayer" is "Yes"

Figure 85: GU class Javadoc, page 2

boolean	**needsServiceLayer**() Returns TRUE if the value of tag "ServiceLayer" is "Yes".
boolean	**needsSessionState**() Returns TRUE iff the value of tag "SessionStateLayer" is "Yes"
Object	**packageIncludes**(Object pkg, String classStr) Checks whether or not the given package includes the class indicated by the given stereotype.
boolean	**patternFound**(String patName) Returns true if the given pattern name is found among the UsedPatternList in th emodel.
boolean	**patternLayerMismatch**(Object pkg) Investigate all the classes in all the subpackages of the package with stereotype *PofEAAModel* and if a class is not located in a right package then returns TRUE as a mismatch.
String	**searchLayerConfigStr**(String stereotype) Returns the name of the layer that must contain the pattern specified by the given stereotype.
Object	**setMultiplicity**(Object Association, String mul1, String mul2) Sets the multiplicity of an association.
void	**setPofEAAPackage**(Object pk) Sets the global variable "pofeaaPkg" as the value given by the parameter.
boolean	**subset**(java.util.Collection set1, java.util.Collection set2) Checks whether or not the first set is a subset of the second set.

Methods inherited from class Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Detail

addAttrToClass

```
public static Object addAttrToClass(Object cls,
                                    String name,
                                    char type)
```

Adds a new attribute, with the name and type specified, to the given class.

Parameters:
`cls` - The class which will contain the attribute
`name` - The name of the attribute
`type` - The type of attribute, currently represented by one character

---

addChildToClass

```
public static Object addChildToClass(Object parent,
                                     String chdName,
                                     String str)
```

Adds a child with given name and stereotype to the given parent.

Parameters:
`parent` - The parent class
`chdName` - The name of the child
`str` - The stereotype of the child, empty "" means no stereotype is added!

Figure 86: GU class Javadoc, page 3

addClientToClass

```
public static Object addClientToClass(Object supplier,
                                      String str)
```

Adds a client with given stereotype to the given class. Note: in A-->B, A is client, B is supplier.

Parameters:
`supplier` - The supplier class, empty "" means no stereotype is added!
`str` - The stereotype of the client

---

addRationale

```
public static void addRationale(String wizardPath)
```

Adds the rationale to the Design Rationale file.

Parameters:
`wizardPath` - The given wizard in the form of a string.

---

addSteToObject

```
public static void addSteToObject(Object obj,
                                  String st)
```

Adds a list of stereotypes to a model element.

Parameters:
`obj` - The model element to which we want to add some stereotypes
`st` - The COMMA separated string of added stereotypes such as "st1,st2,st3"

---

addSupplierToClass

```
public static Object addSupplierToClass(Object client,
                                        String str)
```

Adds a supplier with given stereotype to the given class. Note: in A-->B, A is client, B is supplier.

Parameters:
`client` - The client class
`str` - The stereotype of client, empty "" means no stereotype is added!

---

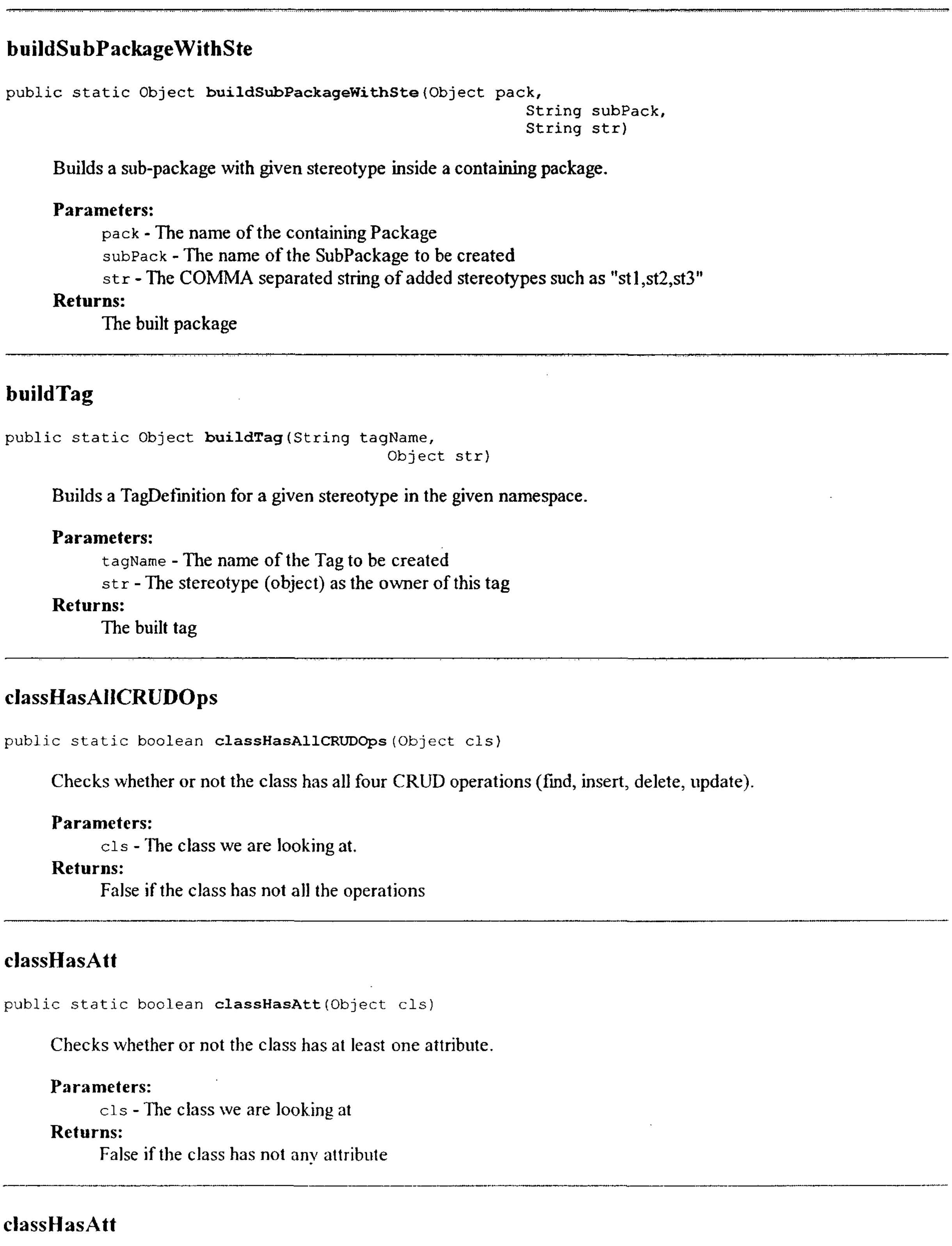
buildOpWithSte

```
public static void buildOpWithSte(Object cls,
                                  String opName,
                                  String str)
```

Builds an operation with given stereotype and adds it to the given class. The return type will be void.

Parameters:
`cls` - The class in which the operation will be created
`opName` - The name of the created operation
`str` - (Maybe Empty) The COMMA separated string of added stereotypes such as "st1,st2,st3"

Figure 87: GU class Javadoc. page 4



buildSubPackageWithSte

```
public static Object buildSubPackageWithSte(Object pack,
                                             String subPack,
                                             String str)
```

Builds a sub-package with given stereotype inside a containing package.

Parameters:
`pack` - The name of the containing Package
`subPack` - The name of the SubPackage to be created
`str` - The COMMA separated string of added stereotypes such as "st1,st2,st3"
Returns:
The built package

---

buildTag

```
public static Object buildTag(String tagName,
                              Object str)
```

Builds a TagDefinition for a given stereotype in the given namespace.

Parameters:
`tagName` - The name of the Tag to be created
`str` - The stereotype (object) as the owner of this tag
Returns:
The built tag

---

classHasAllCRUDOps

```
public static boolean classHasAllCRUDOps(Object cls)
```

Checks whether or not the class has all four CRUD operations (find, insert, delete, update).

Parameters:
`cls` - The class we are looking at.
Returns:
False if the class has not all the operations

---

classHasAtt

```
public static boolean classHasAtt(Object cls)
```

Checks whether or not the class has at least one attribute.

Parameters:
`cls` - The class we are looking at
Returns:
False if the class has not any attribute

---

classHasAtt

Figure 88: GU class Javadoc, page 5

```
public static boolean classHasAtt(Object cls,
                                  String att)
```

Checks whether or not the class has the attribute with given name.

Parameters:
`cls` - The class we are looking at
`att` - The name of the attribute we are looking for
Returns:
True if the required attribute is found

---

classHasOnlyPublicOp

```
public static boolean classHasOnlyPublicOp(Object cls)
```

Checks whether or not all the operations of the class are public.

Parameters:
`cls` - The class we are looking at
Returns:
False if the class has one non-public operation

---

classHasOp

```
public static boolean classHasOp(Object cls)
```

Checks whether or not the class has any operation.

Parameters:
`cls` - The class we are looking at
Returns:
False if the class has not any operation

---

classHasRetOp

```
public static boolean classHasRetOp(Object cls,
                                    String name,
                                    String retType)
```

Checks whether or not the class has an operation with given name and return type.

Parameters:
`cls` - The class we are looking at
`name` - The name of the operation we are looking for
`retType` - The name of returnType we are looking for
Returns:
True if the specified operation is found

---

classHasSteAtt

```
public static boolean classHasSteAtt(Object cls,
                                     String attStr)
```

Figure 89: GU class Javadoc, page 6

Checks whether or not the class has an attribute with the given stereotype.

Parameters:
`cls` - The class we are looking at
`attSt` - The stereotype for attribute we are looking for.
Returns:
True if the required attribute is found

---

classHasSteOp

```
public static boolean classHasSteOp(Object cls,
                                    String opSt)
```

Checks whether or not the class has an operation with the given stereotype.

Parameters:
`opSt` - The operation stereotype
`cls` - The class we are looking at
Returns:
True if the operation is found

---

classHasSteRetOp

```
public static boolean classHasSteRetOp(Object cls,
                                       String opSt,
                                       String retTypeStr)
```

Checks whether or not the class has an operation with given stereotype and with given stereotype for its return type.

Parameters:
`cls` - The class we are looking at
`opSt` - The stereotype of the operation we are looking for
`retTypeStr` - The stereotype of the returnType of the operation
Returns:
true or false

---

classHasStrChild

```
public static Object classHasStrChild(Object cls,
                                      String str)
```

Checks whether or not there is a child for a class with given stereotype.

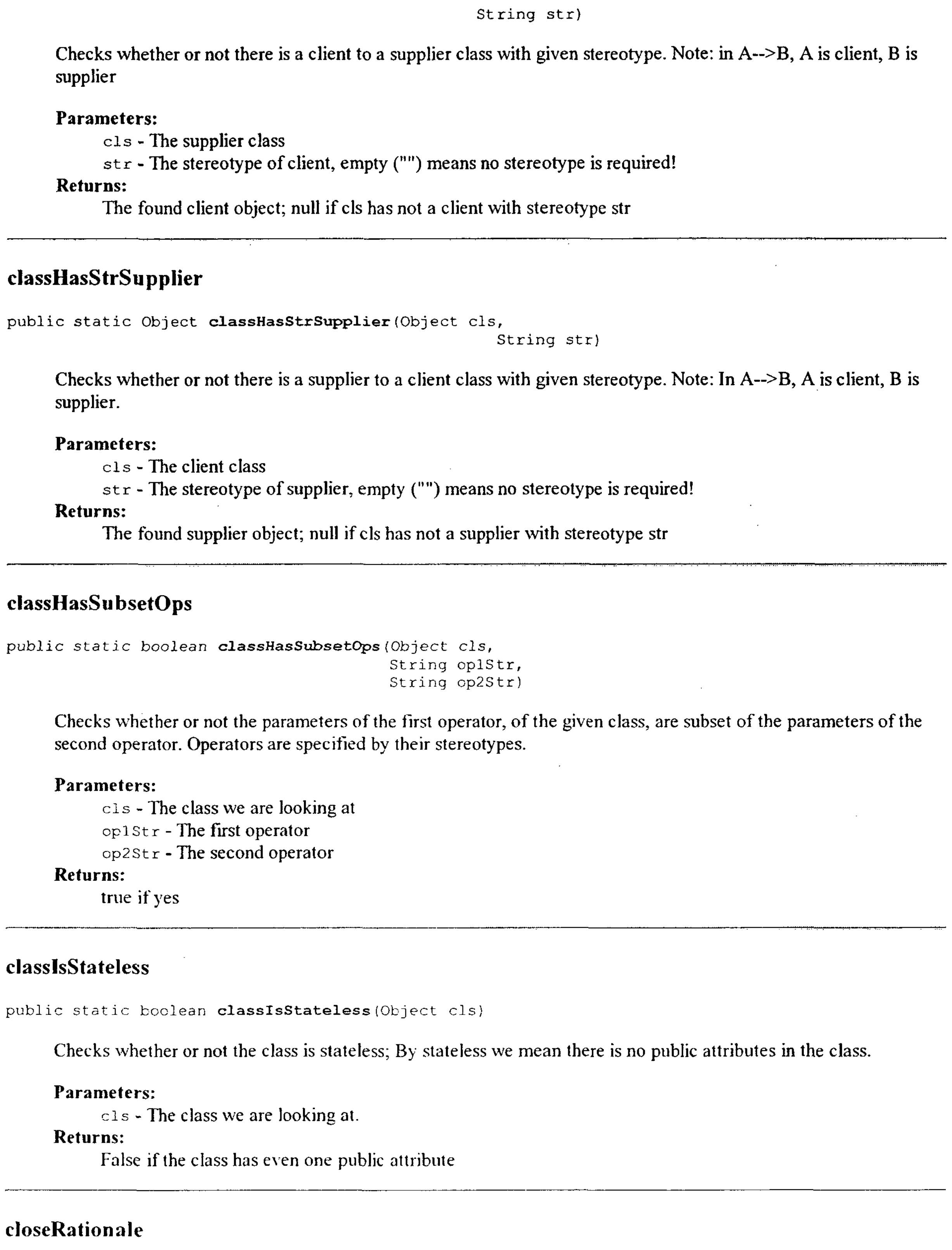
Parameters:
`cls` - The parent class
`str` - The stereotype of child, empty ("") means no stereotype is required! Only one child is enough.
Returns:
The child object, null if cls has not a child with stereotype str

---

classHasStrClient

```
public static Object classHasStrClient(Object cls,
```

Figure 90: GU class Javadoc, page 7



```
                                        String str)
```

Checks whether or not there is a client to a supplier class with given stereotype. Note: in A-->B, A is client, B is supplier

Parameters:
`cls` - The supplier class
`str` - The stereotype of client, empty ("") means no stereotype is required!
Returns:
The found client object; null if cls has not a client with stereotype str

---

classHasStrSupplier

```
public static Object classHasStrSupplier(Object cls,
                                          String str)
```

Checks whether or not there is a supplier to a client class with given stereotype. Note: In A-->B, A is client, B is supplier.

Parameters:
`cls` - The client class
`str` - The stereotype of supplier, empty ("") means no stereotype is required!
Returns:
The found supplier object; null if cls has not a supplier with stereotype str

---

classHasSubsetOps

```
public static boolean classHasSubsetOps(Object cls,
                                        String op1Str,
                                        String op2Str)
```

Checks whether or not the parameters of the first operator, of the given class, are subset of the parameters of the second operator. Operators are specified by their stereotypes.

Parameters:
`cls` - The class we are looking at
`op1Str` - The first operator
`op2Str` - The second operator
Returns:
true if yes

---

classIsStateless

```
public static boolean classIsStateless(Object cls)
```

Checks whether or not the class is stateless; By stateless we mean there is no public attributes in the class.

Parameters:
`cls` - The class we are looking at.
Returns:
False if the class has even one public attribute

---

closeRationale

Figure 91: GU class Javadoc, page 8

```
public static void closeRationale()
```

Closes the Design Rationale file, upon Exit from ArgoUML

---

createRationaleFile

```
public static void createRationaleFile()
```

Creates the Design Rationale for the first time, or for appending.

---

findOp

```
public static Object findOp(Object cls,
                            String opStr)
```

Finds an operator with given stereotype in the given class.

Parameters:
`cls` - The class we are looking at
`opStr` - The stereotype of the operation
Returns:
The found operator object

---

findSteSubPack

```
public static Object findSteSubPack(Object pkg,
                                    String pacStr)
```

Checks whether or not the given package includes the specified stereotyped subpackage.

Parameters:
`pkg` - The package we are looking at
`pacStr` - The stereotype for sub package we are looking for
Returns:
The found subpackage

---

findStrUniCompositionEnd

```
public static Object findStrUniCompositionEnd(Object cls,
                                              String str,
                                              int mlt)
```

Returns the class that is connected to given class by a unidirectional aggregation, if the found class has the requires stereotype and has the specified multiplicity. eg, suppose A --> B. and --> is an AGGREGATION from A to B.

Parameters:
`cls` - The class A
`str` - The stereotype of class B
`mlt` - The multiplicity of the aggregation in B side, -1 for infinity

---

generateType

Figure 92: GU class Javadoc. page 9

`public static Object generateType(char type)`

Builds a type based on given character representative.

Parameters:
`char` - Character representative for the type being created: 'b' for boolean, 'd' for double, 'i' for int, 'f' for float, 'v' for void, 'c' for Currency, 'r' for RecordSet

Returns:
An object for given type

---

getPofeaaPkg

`public static Object getPofeaaPkg()`

returns the object (package) indicating current PofEAA package

---

hasComplexity

`public static boolean hasComplexity(String complexity)`

Returns TRUE if the value of tag "Complexity" is the same as the given parameter. Note: The anticipated values are "Simple", "Moderate", and "Complex". The default is "Simple", ie, if the given parameter is "Simple" and the tag has no value yet, we return TRUE.

Parameters:
`complexity` - The anticipated value for the tag Complexity

---

hasConflict

`public static boolean hasConflict(String conf)`

Returns TRUE if the value of tag "ChanceOfConflict" is the same as the given parameter. Note: The anticipated values are "High" and "Low". The default is "Low", ie, if the given parameter is "Low" and the tag has no value yet, we return TRUE.

Parameters:
`conf` - The anticipated value for the tag Expertise

---

hasExpertise

`public static boolean hasExpertise(String expertise)`

Returns TRUE if the value of tag "Expertise" is the same as the given parameter. Note: The anticipated values are "Novice", "Intermediate", and "Expert". The default is "Novice", ie, if the given parameter is "Novice" and the tag has no value yet, we return TRUE.

Parameters:
`expertise` - The anticipated value for the tag Expertise

---

hasStr

`public static boolean hasStr(Object obj,`

Figure 93: GU class Javadoc, page 10

```
                             String str)
```

Checks whether or not the given element has the specified stereotype among its stereotypes.

Parameters:
`obj` - The model element we are looking at
`str` - The stereotype we are looking for
Returns:
True if the stereotype is found.

---

hasTool

```
public static boolean hasTool(String tool)
```

Returns TRUE if the value of tag "Tool" is the same as the given parameter. Note: default is Java, ie, if the given parameter is "Java" and the tag has no value yet, we return TRUE.

Parameters:
`tool` - The anticipated value for the tag Tool

---

hasViewBuilt

```
public static boolean hasViewBuilt(String viewBuilt)
```

Returns TRUE if the value of tag "ViewBuilt" is the same as the given parameter. Note: The anticipated values are "HTML",and "XSLT". The default is "HTML", ie, if the given parameter is "HTML" and the tag has no value yet, we return TRUE.

Parameters:
`viewBuilt` - The anticipated value for the tag Expertise

---

isDistributed

```
public static boolean isDistributed()
```

Returns TRUE if the value of tag "DistributedLayer" is "Yes"

---

makeElementAbstract

```
public static void makeElementAbstract(Object obj)
```

Makes an element Abstracts.

Parameters:
`obj` - The model element which we want to make it abstract

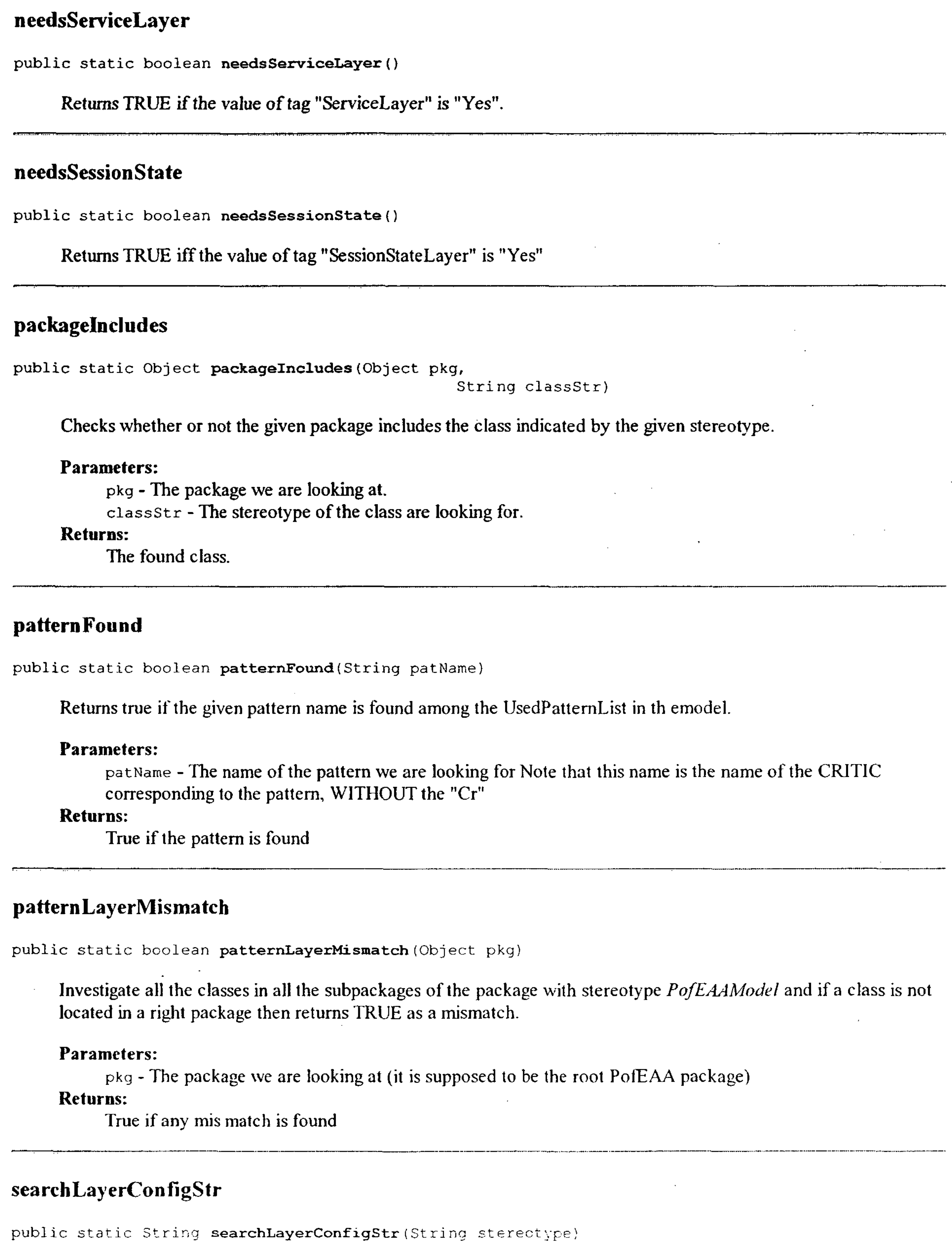
---

needsConcurrency

```
public static boolean needsConcurrency()
```

Returns TRUE iff the value of tag "ConcurrencyLayer" is "Yes"

Figure 94: GU class Javadoc. page 11



needsServiceLayer

```
public static boolean needsServiceLayer()
```

Returns TRUE if the value of tag "ServiceLayer" is "Yes".

---

needsSessionState

```
public static boolean needsSessionState()
```

Returns TRUE iff the value of tag "SessionStateLayer" is "Yes"

---

packageIncludes

```
public static Object packageIncludes(Object pkg,
                                     String classStr)
```

Checks whether or not the given package includes the class indicated by the given stereotype.

Parameters:
`pkg` - The package we are looking at.
`classStr` - The stereotype of the class are looking for.
Returns:
The found class.

---

patternFound

```
public static boolean patternFound(String patName)
```

Returns true if the given pattern name is found among the UsedPatternList in th emodel.

Parameters:
`patName` - The name of the pattern we are looking for Note that this name is the name of the CRITIC corresponding to the pattern, WITHOUT the "Cr"
Returns:
True if the pattern is found

---

patternLayerMismatch

```
public static boolean patternLayerMismatch(Object pkg)
```

Investigate all the classes in all the subpackages of the package with stereotype *PofEAAModel* and if a class is not located in a right package then returns TRUE as a mismatch.

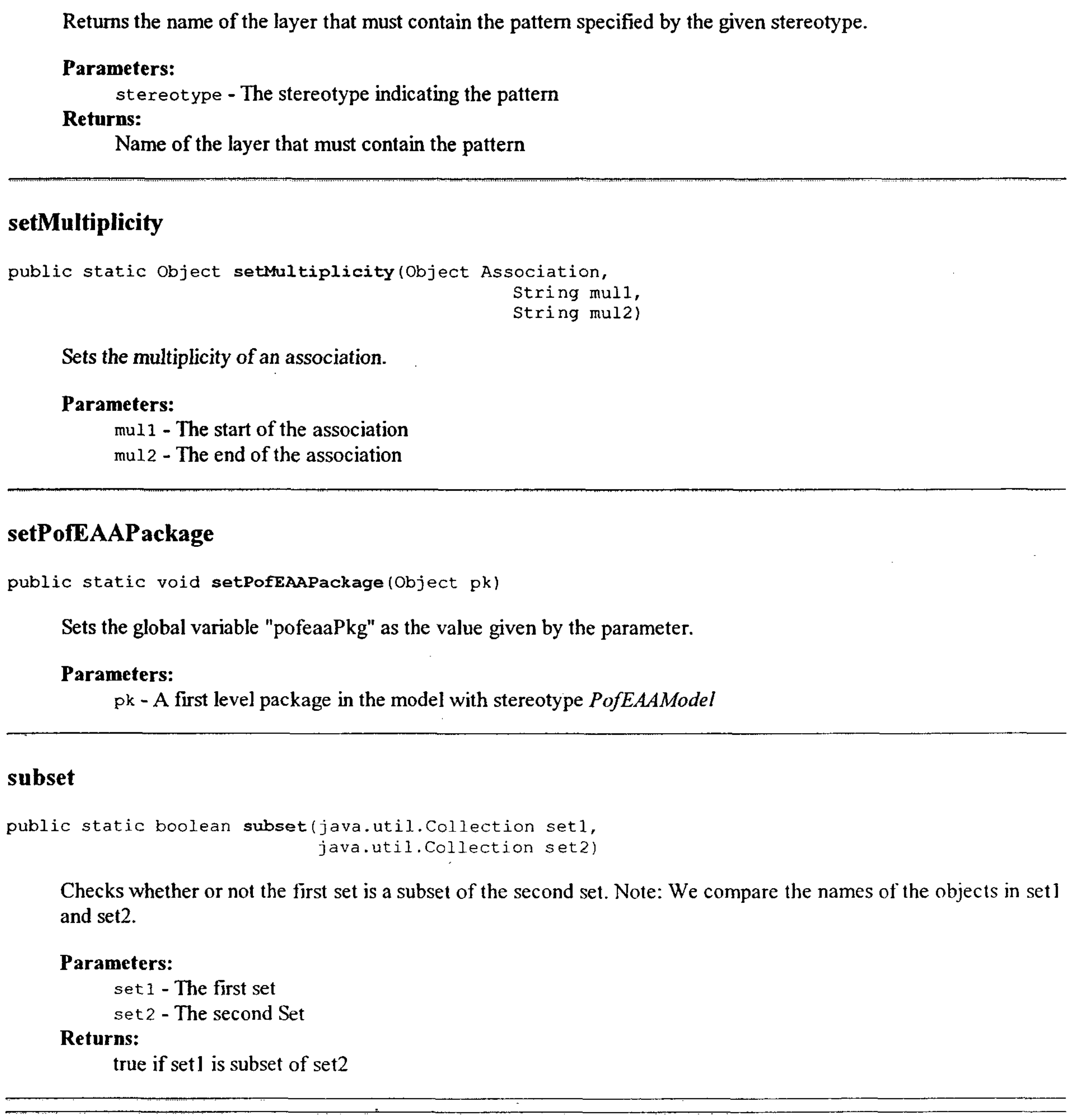
Parameters:
`pkg` - The package we are looking at (it is supposed to be the root PofEAA package)
Returns:
True if any mis match is found

---

searchLayerConfigStr

```
public static String searchLayerConfigStr(String stereotype)
```

Figure 95: GU class Javadoc, page 12



Returns the name of the layer that must contain the pattern specified by the given stereotype.

Parameters:

`stereotype` - The stereotype indicating the pattern

Returns:

Name of the layer that must contain the pattern

---

setMultiplicity

```
public static Object setMultiplicity(Object Association,
                                     String mul1,
                                     String mul2)
```

Sets the multiplicity of an association.

Parameters:

`mul1` - The start of the association

`mul2` - The end of the association

---

setPofEAAPackage

```
public static void setPofEAAPackage(Object pk)
```

Sets the global variable "pofeaaPkg" as the value given by the parameter.

Parameters:

`pk` - A first level package in the model with stereotype *PofEAAModel*

---

subset

```
public static boolean subset(java.util.Collection set1,
                             java.util.Collection set2)
```

Checks whether or not the first set is a subset of the second set. Note: We compare the names of the objects in set1 and set2.

Parameters:

`set1` - The first set

`set2` - The second Set

Returns:

true if set1 is subset of set2

Figure 96: GU class Javadoc, page 13

A.5.2 A Structural Critic

FrontController.java

```
package org.argouml.pattern.cognitive.PofEAA.critics;

import java.util.Collection;
import java.util.Iterator;
import org.argouml.cognitive.Critic;
import org.argouml.cognitive.Designer;
import org.argouml.cognitive.ToDoItem;
import org.argouml.model.Model;
import org.argouml.pattern.cognitive.PofEAA.util.GU;
import org.argouml.pattern.cognitive.PofEAA.wizards.WizFrontController;
import org.argouml.uml.cognitive.UMLDecision;
import org.argouml.uml.cognitive.critics.CrUML;
/**
 * This is a critic to find Fowler's Front Controller pattern.
 * Following are the requirements for detecting this pattern:
 * <ul>
 * <li> A class with stereotype <em>handler</em>.
 * <li> Handler class must have two operations with stereotypes
 *      <em>doget</em> and <em>dopost</em>.
 * <li> Handler class should be a client of a supplier class with stereotypes
 *      <em>command</em>.
 * <li> Command class should have operations with stereotype <em>process</em>.
 * <li> Command class should have child classes with stereotype <em>concretecommand</em>.
 * <li> ConcreteCommand class should have operations with stereotype <em>process</em>.
 * </ul>
 * @see PofEAA book, P. 344 (PLV Rule (advice): A25).
 * @version Structural
 * @author Bahman Zamani & Sahar Kayhani
 */
public class CrFrontController extends CrUML{
  public CrFrontController(){
    setupHeadAndDesc();
    addSupportedDecision(UMLDecision.PATTERNSOFEAA);
    setKnowledgeTypes(Critic.KT_POFEAASTR);
    setPriority(ToDoItem.HIGH_PRIORITY);
  }
  public boolean predicate(Object dm, Designer dsgr) {
   if (dm == null) return NO_PROBLEM;
   if (! Model.getFacade().isAClass(dm)) return NO_PROBLEM;
   Object aClass = dm;
   // aClass should have stereotype <<frontController>>, this is the sign
   // of pattern that is applied on the Handler class
   if (! GU.objectHasSte(aClass, "frontController"))    /*SIGN*/
      return NO_PROBLEM;
   // class should have at least one operation
   if (! GU.classHasOp(aClass) ) return PROBLEM_FOUND;
   // Both doGet and doPost ops are required
   boolean getOperationExist  = GU.classHasSteOp(aClass,"doget");
   if (! getOperationExist ) return PROBLEM_FOUND;
   boolean postOperationExist = GU.classHasSteOp(aClass,"dopost");
   if (! postOperationExist ) return PROBLEM_FOUND;
   // Check if there is a client
   Collection depSet = Model.getFacade().getClientDependencies(aClass);
   if ( depSet.isEmpty() ) return PROBLEM_FOUND;
   // at least one of the suppliers should have the COMMAND structure
```

```
    // An ABSTRACT class with stereotype <<command>> and  one <<process>>
    // operation as well as at least one child with stereotype
    // <<concreteCommand>> and with one <<process>> operation
    boolean supplierFound = false;
    Iterator deps = depSet.iterator();
    while ( deps.hasNext() ) {
      Object dep = deps.next();
     Collection supplierSet = Model.getFacade().getSuppliers(dep);
      if ( supplierSet.isEmpty() ) continue;
      Iterator suppliers = supplierSet.iterator();
      while ( suppliers.hasNext() && !supplierFound) {
        // This should be the Command class
        Object supplier = suppliers.next();
        if ( GU.objectHasSte(supplier, "command") ) {
          if (Model.getFacade().isAbstract(supplier)) {
            if (GU.classHasSteOp(supplier,"process")) {
              // We need at least one child which is CONCRETE
              // and has PROCESS operation
              Collection children = Model.getFacade().getChildren(supplier);
              if ((children.isEmpty())) return PROBLEM_FOUND;
              Iterator child = children.iterator();
              while (child.hasNext()) {
                Object conCommand = child.next();
                // concrete command must be a class
                if (! Model.getFacade().isAClass(conCommand)) continue;
                //  concrete command class  must be concrete
                if (Model.getFacade().isAbstract(conCommand)) continue;
                if (!GU.objectHasSte(conCommand,"concretecommand"))
                  return PROBLEM_FOUND;
                if (!GU.classHasSteOp(conCommand,"process"))
                  return PROBLEM_FOUND;
                // Now, report the correct usage of FC pattern
                classNames.add(Model.getFacade().getName(conCommand)+"
                -> concrete command");
              }
              supplierFound = true;
              classNames.add(Model.getFacade().getName(supplier)+"->command");
    } } } } }
    if ( ! supplierFound ) return PROBLEM_FOUND;
    PATTERN_FOUND = true;
    classNames.add(Model.getFacade().getName(aClass)+" -> handler");
      patternLayer = Model.getFacade().getName(Model.getFacade().getNamespace(aClass));
    return NO_PROBLEM;
  }
  public Class getWizardClass(ToDoItem item) {
    return WizFrontController.class;
  }
}
```

A.5.3 Three Syntactic Critics

CrLayers.java

```
package org.argouml.pattern.cognitive.PofEAA.critics;

import java.util.Iterator;
import org.argouml.cognitive.Critic;
import org.argouml.cognitive.Designer;
import org.argouml.cognitive.ToDoItem;
import org.argouml.model.Model;
import org.argouml.pattern.cognitive.PofEAA.util.GU;
import org.argouml.pattern.cognitive.PofEAA.wizards.WizLayers;
import org.argouml.uml.cognitive.UMLDecision;
import org.argouml.uml.cognitive.critics.CrUML;

public class CrLayers extends CrUML {
  public CrLayers(){
    setupHeadAndDesc();
    addSupportedDecision(UMLDecision.PATTERNSOFEAA);
    setKnowledgeTypes(Critic.KT_POFEAASYN);
    setPriority(ToDoItem.HIGH_PRIORITY);
  }

  public boolean predicate2(Object dm, Designer dsgr) {
    if (dm == null) return NO_PROBLEM;
    if (Model.getFacade().isAModel(dm)) return NO_PROBLEM;
    if (!(Model.getFacade().isAPackage(dm))) return NO_PROBLEM;
    Object aPackage = dm;
    if (!GU.hasStr(aPackage, "PofEAAModel"))  return NO_PROBLEM;
    GU.setPofEAAPackage(aPackage);

    boolean presentationFound   = false; boolean serviceFound       = false;
    boolean domainFound         = false; boolean dataSourceFound    = false;
    boolean basicFound          = false; boolean distributedFound   = false;
    boolean concurrencyFound    = false; boolean sessionStateFound  = false;
    String prs = ""; String srv = ""; String dom = ""; String ds = "";
    String bas = ""; String dis = "";  String conc= ""; String ses = "";
    Iterator innerElms = Model.getFacade().getOwnedElements(aPackage).iterator();
    while (innerElms.hasNext()) {
        Object elmnt = innerElms.next();
        if ( elmnt != null ) {
            if (Model.getFacade().isAPackage(elmnt)) {
                if (GU.hasStr(elmnt, "presentation")) {
                   presentationFound = true;
                   prs = Model.getFacade().getName(elmnt);
                }
                else if (GU.hasStr(elmnt, "service")) {
                    serviceFound = true;
                    srv = Model.getFacade().getName(elmnt);
                }
                else if (GU.hasStr(elmnt, "domain")) {
                    domainFound = true;
                    dom = Model.getFacade().getName(elmnt);
                }
                else if (GU.hasStr(elmnt, "datasource")) {
                    dataSourceFound = true;
                    ds = Model.getFacade().getName(elmnt);
                }
                else if (GU.hasStr(elmnt, "basic")) {
                    basicFound = true;
                    bas = Model.getFacade().getName(elmnt);
                }
                else if (GU.hasStr(elmnt, "distributed")) {
                    distributedFound = true;
                    dis = Model.getFacade().getName(elmnt);
                }
                else if (GU.hasStr(elmnt, "concurrency")) {
                    concurrencyFound = true;
                    conc = Model.getFacade().getName(elmnt);
                }
                else if (GU.hasStr(elmnt, "sessionstate")) {
                    sessionStateFound = true;
                    ses = Model.getFacade().getName(elmnt);
                }
    }   }   }

    if ( (!serviceFound      && GU.needsServiceLayer())       ||
         (!distributedFound  && GU.needsDistributedLayer() ) ||
         (!concurrencyFound  && GU.needsConcurrencyLayer() ) ||
         (!sessionStateFound && GU.needsSessionStateLayer()) ||
         !presentationFound || !domainFound || !dataSourceFound || !basicFound ) {
        return PROBLEM_FOUND;
    }

    PATTERN_FOUND = true;
    classNames.add(prs+" -> Presentation Layer Package");
    if (serviceFound) classNames.add(srv+" -> Service Layer Package");
    classNames.add(dom+" -> Domain layer Package");
    classNames.add(ds+ " -> Data Source Layer Package");
    classNames.add(bas+" -> Basic Layer Package");
    if (distributedFound) classNames.add(dis+" -> Distributed Package");
    if (concurrencyFound) classNames.add(conc+" -> Concurrency Package");
    if (sessionStateFound) classNames.add(ses+" -> Session State Package");
    classNames.add(Model.getFacade().getName(aPackage)+" -> PofEAA Model");
    patternLayer = Model.getFacade().getName(Model.getFacade().getNamespace(aPackage));
    return NO_PROBLEM;

  }
  public Class getWizardClass(ToDoItem item) {
     return WizLayers.class;
  }

}
```

CrPatterns.java

```
package org.argouml.pattern.cognitive.PofEAA.critics;
import org.argouml.cognitive.Critic;
import org.argouml.cognitive.Designer;
import org.argouml.cognitive.ToDoItem;
import org.argouml.model.Model;
import org.argouml.pattern.cognitive.PofEAA.util.GU;
import org.argouml.pattern.cognitive.PofEAA.wizards.WizPatterns;
import org.argouml.uml.cognitive.UMLDecision;
import org.argouml.uml.cognitive.critics.CrUML;

/**
 * This critic checks whether the patterns are placed in correct layers or not?
 * In case of any mismatch between the layer that includes the pattern, and the
 * anticipated layer, a syntax error is reported.
 * @see PofEAA book, http://martinfowler.com/eaaCatalog/
 * @version Syntactic
 * @author Bahman Zamani (with contributions by Sahar Kayhani)
 */
public class CrPatterns extends CrUML {
    public CrLayers2(){
        setupHeadAndDesc();
        addSupportedDecision(UMLDecision.PATTERNSOFEAA);
        setKnowledgeTypes(Critic.KT_POFEAASYN);
        setPriority(ToDoItem.HIGH_PRIORITY);
    }

    public boolean predicate2(Object dm, Designer dsgr) {
        if (dm == null) return NO_PROBLEM;
        // Note that the root model is also a package!
        // Do not apply this critic on that!
        if (Model.getFacade().isAModel(dm)) return NO_PROBLEM;
        if (!(Model.getFacade().isAPackage(dm))) return NO_PROBLEM;
        Object aPackage = dm;
        // Only look inside the main package which is a package with
        // stereotype <<PofEAAModel>>
        if (!GU.hasStr(aPackage, "PofEAAModel")) return NO_PROBLEM;
        // If any mismatch is found between a pattern and its containing layer,
        // then trigger this critic
        if (!GU.patternLayerMismatch (aPackage)) return NO_PROBLEM;
        return PROBLEM_FOUND;
    }
    public Class getWizardClass(ToDoItem item) {
        return WizPatterns.class;
    }
}
```

CrDomainModelSyn.java

```
package org.argouml.pattern.cognitive.PofEAA.critics;

import org.argouml.cognitive.Critic;
import org.argouml.cognitive.Designer;
import org.argouml.cognitive.ToDoItem;
import org.argouml.model.Model;
import org.argouml.pattern.cognitive.PofEAA.util.GU;
import org.argouml.pattern.cognitive.PofEAA.wizards.WizDomainModelSyn;
import org.argouml.uml.cognitive.UMLDecision;
import org.argouml.uml.cognitive.critics.CrUML;
/**
 * This is a critic to find syntactic errors regarding the relationship between the
 * Domain Model pattern and the patterns in the Data Source Layer.
 * Following are the requirements for detecting such errors.
 * <ul>
 * <li> A pattern <em>DomainModel</em> is already detected and is in the Domain Layer.
 * <li> Either state A or B happened.
 * <ul>
 * <li> A: A dependency is found from <em>DomainModel</em> to a pattern
 * <li> <em>ActiveRecord</em> which is already detected and is in the Data Source Layer.
 * <li> The model is not Simple.
 * </ul>
 * <ul>
 * <li> B: A dependency is found from <em>DomainModel</em> to a pattern
 * <li> <em>DataMapper</em> which is already detected and is in the Data Source Layer.
 * <li> The model is not Complex.
 * <ul>
 * </ul>
 * @see PofEAA book, P.36,117   (PLV Rule: R12,R13,R14).
 * @version Syntactic
 * @author Bahman Zamani, 20 Nov 2008  , 6 Apr 09
 */
public class CrDomainModelSyn extends CrUML {

    public CrDomainModelSyn(){
        setupHeadAndDesc();
        addSupportedDecision(UMLDecision.PATTERNSOFEAA);
        setKnowledgeTypes(Critic.KT_POFEAASYN);
        setPriority(ToDoItem.HIGH_PRIORITY);
    }
    public boolean predicate2(Object dm, Designer dsgr) {
        if (dm == null) return NO_PROBLEM;
        if (! Model.getFacade().isAClass(dm)) return NO_PROBLEM;
        Object dmCls = dm;
        if (!GU.hasStr(dmCls, "DomainModel")) return NO_PROBLEM;
        // Pattern must be already found
        if (!GU.patternFound("DomainModel")) return NO_PROBLEM;
        // DomainModel Pattern must be in the correct Layer
        Object dmPkg = Model.getFacade().getNamespace(dmCls);
        if ( !GU.hasStr(dmPkg, "domain") ) return NO_PROBLEM;

        // If DomainModel uses Active Record
        Object actRec = GU.findStrSupplier(dmCls, "ActiveRecord");
        if ( actRec != null ) {
             // ActiveRecord Pattern must be already found
             if ( GU.patternFound("ActiveRecord")) {
                 // ActiveRecord Pattern must be in the correct Layer
                 Object dsPkg = Model.getFacade().getNamespace(actRec);
                 if ( GU.hasStr(dsPkg, "dataSource") )
                     // If the model is not Simple, it is a sign of an error
                     if ( ! GU.hasComplexity("Simple") ) return PROBLEM_FOUND;
             }
        }
        // If DomainModel uses Data Mapper
        Object dataMap = GU.findStrSupplier(dmCls, "DataMapper");
        if ( dataMap != null ) {
             // DataMapper Pattern must be already found
             if ( GU.patternFound("DataMapper")) {
                 // DataMapper Pattern must be in the correct Layer
                 Object dsPkg = Model.getFacade().getNamespace(dataMap);
                 if (GU.hasStr(dsPkg, "dataSource"))
                     // If the model is not Complex, it is a sign of an error
                     if ( ! GU.hasComplexity("Complex") ) return PROBLEM_FOUND;
             }
        }
        return NO_PROBLEM;
    }
    public Class getWizardClass(ToDoItem item) {
        return WizDomainModelSyn.class;
    }
}
```

A.5.4 A Semantic Critic

CrTableDataGatewaySem.java

```
package org.argouml.pattern.cognitive.PofEAA.critics;

import org.argouml.cognitive.Critic;
import org.argouml.cognitive.Designer;
import org.argouml.cognitive.ToDoItem;
import org.argouml.model.Model;
import org.argouml.pattern.cognitive.PofEAA.util.GU;
import org.argouml.pattern.cognitive.PofEAA.wizards.WizTableDataGatewaySem;
import org.argouml.uml.cognitive.UMLDecision;
import org.argouml.uml.cognitive.critics.CrUML;
/**
 * This is a critic to find semantic errors in Fowler's Table Data Gateway pattern.
 * Following are the requirements for detecting the error:
 * <ul>
 * <li> A class with stereotype <em>TableDataGateway</em>
 * <li> pattern is already detected and reported in the PIT
 * <li> The parameter list of insert() should be a subset of parameters of update()
 * </ul>
 *
 * @see PofEAA book, P. 144 (PLV Rule: R46).
 * @version Semantic
 * @author Bahman Zamani
 */
public class CrTableDataGatewaySem extends CrUML {

  public CrTableDataGatewaySem() {
    setupHeadAndDesc();
    addSupportedDecision(UMLDecision.PATTERNSOFEAA);
    setKnowledgeTypes(Critic.KT_POFEAASEM);
    setPriority(ToDoItem.HIGH_PRIORITY);
  }

  public boolean predicate2(Object dm, Designer dsgr) {
    if (dm == null) return NO_PROBLEM;
    if (! Model.getFacade().isAClass(dm)) return NO_PROBLEM;
    Object aClass = dm;
    // aClass should have stereotype <<TableDataGateway>>
    if (!GU.hasStr(aClass, "TableDataGateway"))  return NO_PROBLEM;
    // The TDG pattern should be already detected and recorded in PIT classNames
    if (! GU.patternFound("TableDataGateway"))  return NO_PROBLEM;
    // update() should contain all parameters of insert()
    // Normally, update needs an ID or Key as extra parameter
    if (! GU.classHasSubsetOps(aClass, "insert", "update")) return PROBLEM_FOUND;
   return NO_PROBLEM;
  }
  public Class getWizardClass(ToDoItem item) {
    return WizTableDataGatewaySem.class;
  }
}
```

A.5.5 A Wizard

WizFrontController.java

```
package org.argouml.pattern.cognitive.PofEAA.wizards;

import java.util.Collection;
import java.util.Iterator;
import java.util.Vector;
import javax.swing.JPanel;
import org.apache.log4j.Logger;
import org.argouml.cognitive.ui.WizStepChoice;
import org.argouml.i18n.Translator;
import org.argouml.model.Model;
import org.argouml.pattern.cognitive.PofEAA.util.GU;
import org.argouml.uml.cognitive.critics.UMLWizard;
/**
 * Wizard class for CrFrontController critic.
 * This wizard helps user to add missing <em>doGet</em> or <em>doPost</em> operations
 * to the Handler class.
 * Also to add supplier class with stereotype <em>Command</em> to the Handler class.
 * Also to add <em>process</em> operation in the Command class.
 * Also to make the Command class, abstract.
 * Also to add <em>ConcreteCommand</em> children to the Command class.
 * Also to add <em>process</em> operation in the Concrete Command class.
 *
 * @author Bahman Zamani 13 Aug 2008
 * (With contributions by Sahar Kayhani)
 */
public class WizFrontController extends UMLWizard {
  // Bahman Zamani - 19 Aug 2008
  // We need to record which class is the Command class and which one is concreteCommmand
  // This way in doAction method, it's easy to add missing items to it
  private Object commandClass = null;
  private Object conCommandClass = null;
  private WizStepChoice step1Choice = null;
  private String[] missingItems = new String[5];
  private int missItemCounter = 0;
  private Object triggerClass = null;
  private String instructions = Translator.localize("critics.WizFrontControllert-ins");
  private static final Logger LOG = Logger.getLogger(WizFrontController.class);
  public WizFrontController () {}
  private Object getTriggerClass() {
    if ((triggerClass == null) && (getToDoItem() != null)) {
        triggerClass = getModelElement();
    }
    return triggerClass;
  }
  private Vector buildOptions() {
    Object cls = getTriggerClass();
    if (cls == null)
        return null;
    Vector res = new Vector();
    if (!GU.classHasSteOp(cls,"doGet"))  {
      res.addElement(Translator.localize("critics.WizFrontController-option1"));
      missingItems[missItemCounter++] = "doGet";
    }
    if (!GU.classHasSteOp(cls,"doPost"))  {
      res.addElement(Translator.localize("critics.WizFrontController-option2"));
      missingItems[missItemCounter++] = "doPost";
```

```
}
// We need a <<Command>> supplier class
Collection depSet = Model.getFacade().getClientDependencies(cls);
if( depSet.isEmpty()){
  res.addElement(Translator.localize("critics.WizFrontController-option3"));
  missingItems[missItemCounter++] = "Command";
}
else {
  boolean commandFound = false;
  Iterator deps = depSet.iterator();
  while ( deps.hasNext() && !commandFound) {
    Object dep = deps.next();
    Collection supplierSet = Model.getFacade().getSuppliers(dep);
    if ( supplierSet.isEmpty() ) {
        continue;
    }
    Iterator suppliers = supplierSet.iterator();
    while ( suppliers.hasNext() && !commandFound) {
        commandClass = suppliers.next();
        if ( GU.hasStr(commandClass, "Command") ) {
            commandFound = true;
        }
    }
  }
  if ( ! commandFound ){
    res.addElement(Translator.localize("critics.WizFrontController-option3"));
    missingItems[missItemCounter++] = "Command";
  }
  else {
    // <<Command>> class should be Abstract
    if (! Model.getFacade().isAbstract(commandClass)) {
      res.addElement(Translator.localize("critics.WizFrontController-option4"));
      missingItems[missItemCounter++] = "commandAbs";
    }
    // <<Command>> class needs <<process>> operation
    if (! GU.classHasSteOp(commandClass,"process")) {
      res.addElement(Translator.localize("critics.WizFrontController-option5"));
      missingItems[missItemCounter++] = "commandProcess";
    }
    // <<Command>> class needs at least one child
    Collection children = Model.getFacade().getChildren(commandClass);
    if ( children.isEmpty()) {
      res.addElement(Translator.localize("critics.WizFrontController-option6"));
      missingItems[missItemCounter++] = "commandChildren";
    }
    else {
      // All children need <<ConcreteCommand>> stereotype
      boolean conCommandFound = false;
      Iterator child = children.iterator();
      while (child.hasNext()) {
        conCommandClass = child.next();
        if ( GU.hasStr(conCommandClass,"ConcreteCommand")) {
          conCommandFound = true;
          // each child needs <<process>> operation
          if (! GU.classHasSteOp(conCommandClass,"process")) {
            res.addElement(Translator.localize("critics.WizFrontController-option7"));
            missingItems[missItemCounter++] = "conCommandProcess";
          }
        }
```

```
          // missing stereotype,  if there is only one child without stereotype,
          // this will cause DUPLICATE wizard options but no problem!
          else {
            res.addElement(Translator.localize("critics.WizFrontController-option8"));
            missingItems[missItemCounter++] = "conCommandSte";
          }
        }
        // Not seeing <<concreteCommand>> at all
        if ( ! conCommandFound ) {
          res.addElement(Translator.localize("critics.WizFrontController-option6"));
          missingItems[missItemCounter++] = "commandChildren";
        }

      }
    }
  }
  // If there is more than one option, give an option for selecting all items
  if (missItemCounter>1)
    res.addElement ( Translator.localize("critics.WizFrontController-option9"));
  return res;
}
/**
 * Set the initial instruction string for the choice. May be
 * called by the creator of the wizard to override the default.<p>
 *
 * @param s The new instructions.
 */
public void setInstructions(String s) {
    instructions = s;
}
public JPanel makePanel(int newStep) {
  switch (newStep) {
  case 1:
    if (step1Choice == null) {
      Vector opts = buildOptions();
      if (opts != null) {
        step1Choice = new WizStepChoice(this, instructions, opts);
        step1Choice.setTarget(getToDoItem());
      }
    }
    return step1Choice;
  default:
  }
  return null;
}
// Bahman Zamani - 19 Aug 2008: prevent duplicate creating of model elements
boolean missingCommandCreated = false;
boolean missingConCommandCreated = false;
@Override
public void doAction(int oldStep) {
  switch (oldStep) {
  case 1:
    int choice = -1;
    if (step1Choice != null) choice = step1Choice.getSelectedIndex();
    if (choice == -1) {
      LOG.warn("WizFrontController: nothing selected,  should not get here");
      return;
    }
    try {
        Object handlerClass = getTriggerClass(); // It's the Handler class
        Object curPackage = Model.getFacade().getNamespace(handlerClass);

        // if user has selected to create all missing operations
        if (choice == missItemCounter) {
          for (int i=0; i<choice; i++)
            fixFCProblems(handlerClass, curPackage, i);
        }
        // create operations one by one
        else
          fixFCProblems(handlerClass, curPackage, choice);
      }
      catch(Exception e){
          LOG.error("WizFrontController: could not set operation.", e);
      }
    default:
    }
  }
/**
 * Fixes the problems found in the FrontController pattern
 * @param handlerClass The Handler class in FrontController pattern
 * @param curPackage The current package containing the FrontController pattern
 * @param n The number in missingItem list
 * @author Bahaman Zamani
 */
private void fixFCProblems(Object handlerClass, Object curPackage, int n) {
    // We build doGet and doPost ops in the Handler class
    if ( missingItems[n].equals("doGet") || missingItems[n].equals("doPost") ) {
      if ( ! GU.classHasSteOp(handlerClass, missingItems[n])) {
        GU.buildOpWithSte(handlerClass, missingItems[n]+"Op", missingItems[n]);
      }
    }
    // We build a Command hierarchy and process operations
    else if ( missingItems[n].equals("Command") ) {
      if ( ! missingCommandCreated ) {
        Object newCommandClass = Model.getCoreFactory().
         buildClass("CommandCls",curPackage);
        Model.getCoreFactory().buildDependency(handlerClass,newCommandClass);
        GU.addSteToObject(newCommandClass, "Command");
        // change Command class to Abstract
        GU.makeElementAbstract(newCommandClass);
        Object conCommandClass = Model.getCoreFactory().
         buildClass("ConcreteCommandCls",curPackage);
        GU.addSteToObject(conCommandClass, "ConcreteCommand");
        Model.getCoreFactory().buildGeneralization(conCommandClass,newCommandClass);
        GU.buildOpWithSte(newCommandClass, "processOp","process");
        GU.buildOpWithSte(conCommandClass, "processOp","process");
        missingCommandCreated = true;
      }
    }
    else if( missingItems[n].equals("commandAbs") ) {
      GU.makeElementAbstract(commandClass);
    }
    else if ( missingItems[n].equals("commandProcess") ) {
      if ( ! GU.classHasSteOp(commandClass, "process")) {
          GU.buildOpWithSte(commandClass, "processOp","process");
      }
    }
    else if ( missingItems[n].equals("commandChildren") ) {
      if ( ! missingConCommandCreated ) {
        Object conCommandClass = Model.getCoreFactory().
         buildClass("ConcreteCommand",curPackage);
        GU.addSteToObject(conCommandClass, "ConcreteCommand");
        Model.getCoreFactory().buildGeneralization(conCommandClass,commandClass);
        if ( ! GU.classHasSteOp(conCommandClass, "process")) {
            GU.buildOpWithSte(conCommandClass, "processOp","process");
        }
        missingConCommandCreated = true;
      }
    }
    else if ( missingItems[n].equals("conCommandProcess") ) {
      if ( ! GU.classHasSteOp(conCommandClass, "process")) {
        GU.buildOpWithSte(conCommandClass, "processOp","process");
      }
    }
    else if ( missingItems[n].equals("conCommandSte") ) {
      GU.addSteToObject(conCommandClass, "ConcreteCommand");
    }
  }
}
```

A.6 Sample Application: Online Student Registration System

A.6.1 Domain Model of the System

Figure 97 shows the domain model of the *Online Student Registration System.*

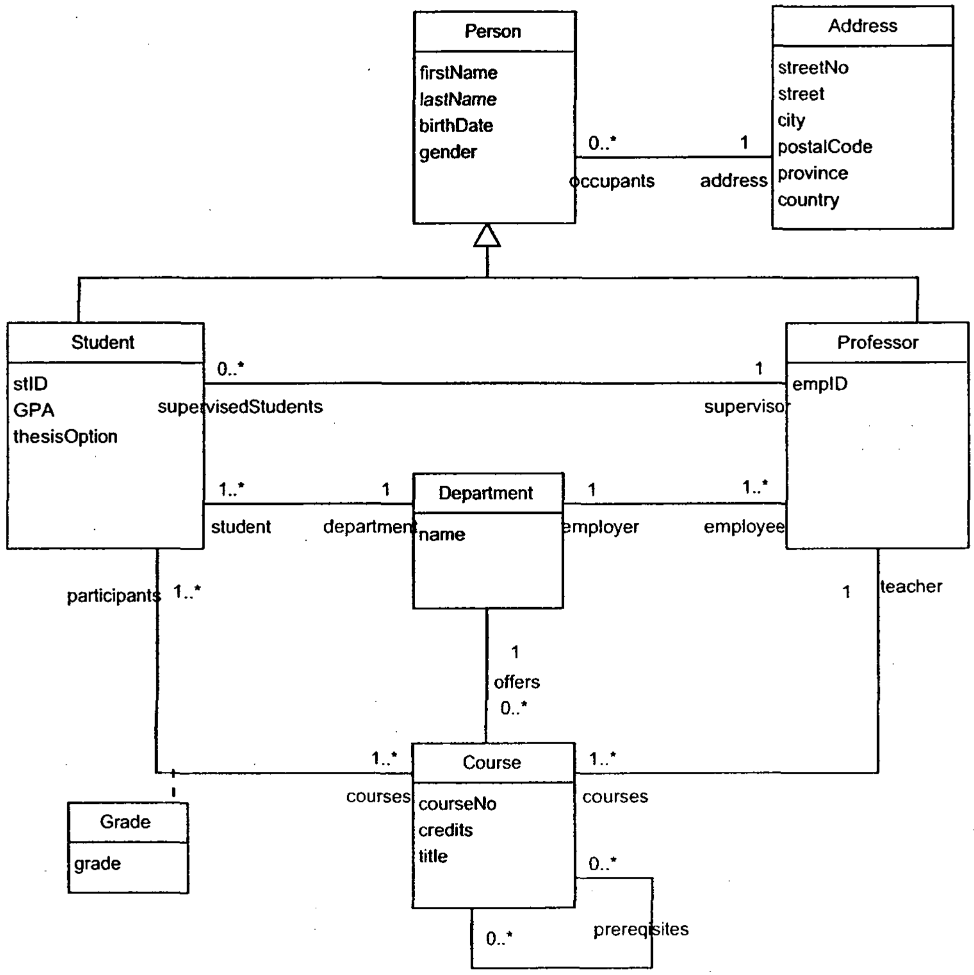


Figure 97: Domain Model of the Online Student Registration System

A.6.2 A Given Design of the System using *PofEAA* Patterns

Figure 98 shows a design of the *Online Student Registration System* using the *PofEAA* patterns.

A.6.3 The Given Design after Verification by the ArgoPLV

Figure 99 shows the given design after it is verified by the ArgoPLV.

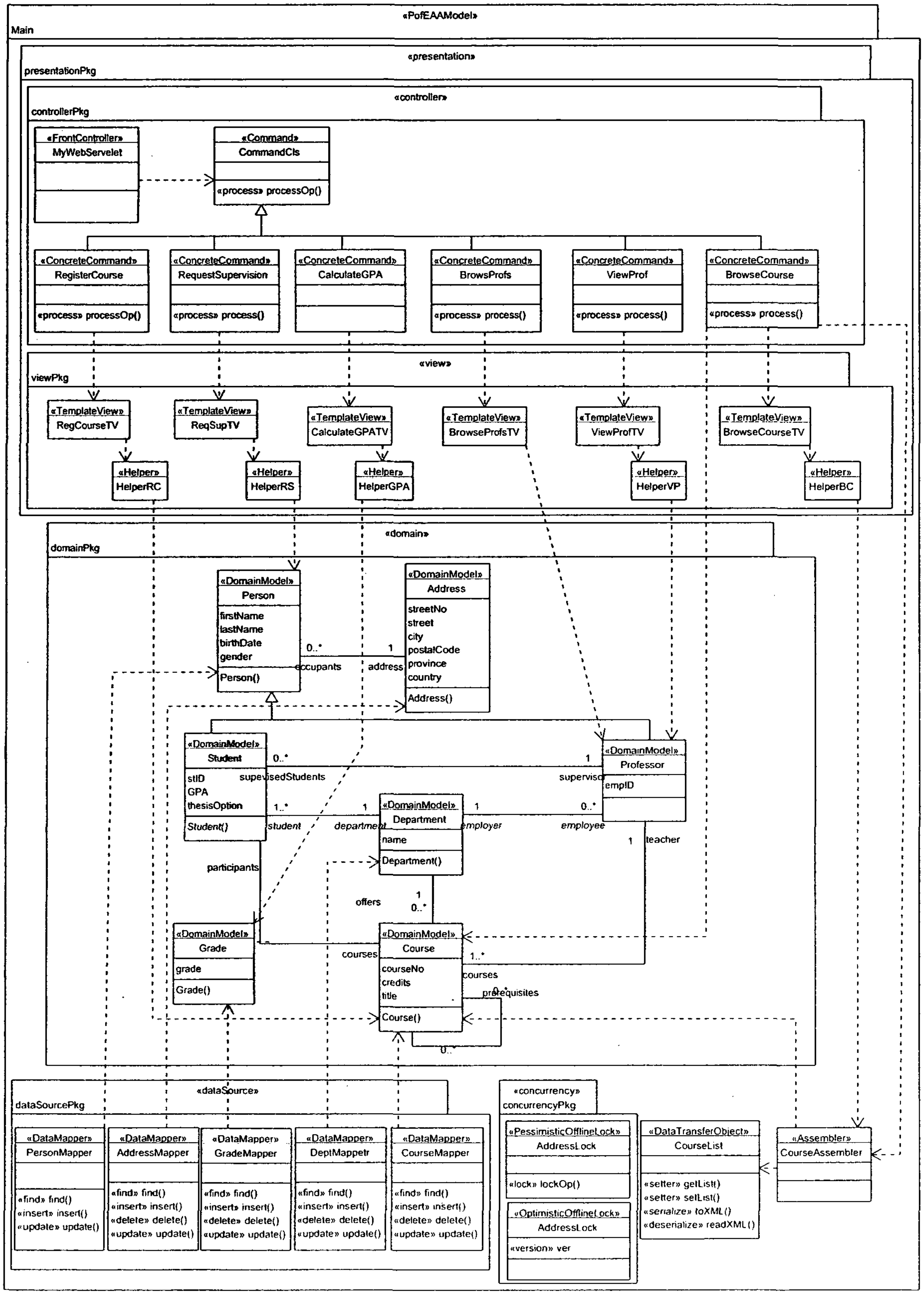


Figure 98: A Design Model for Online Student Registration System using *PofEAA* Patterns

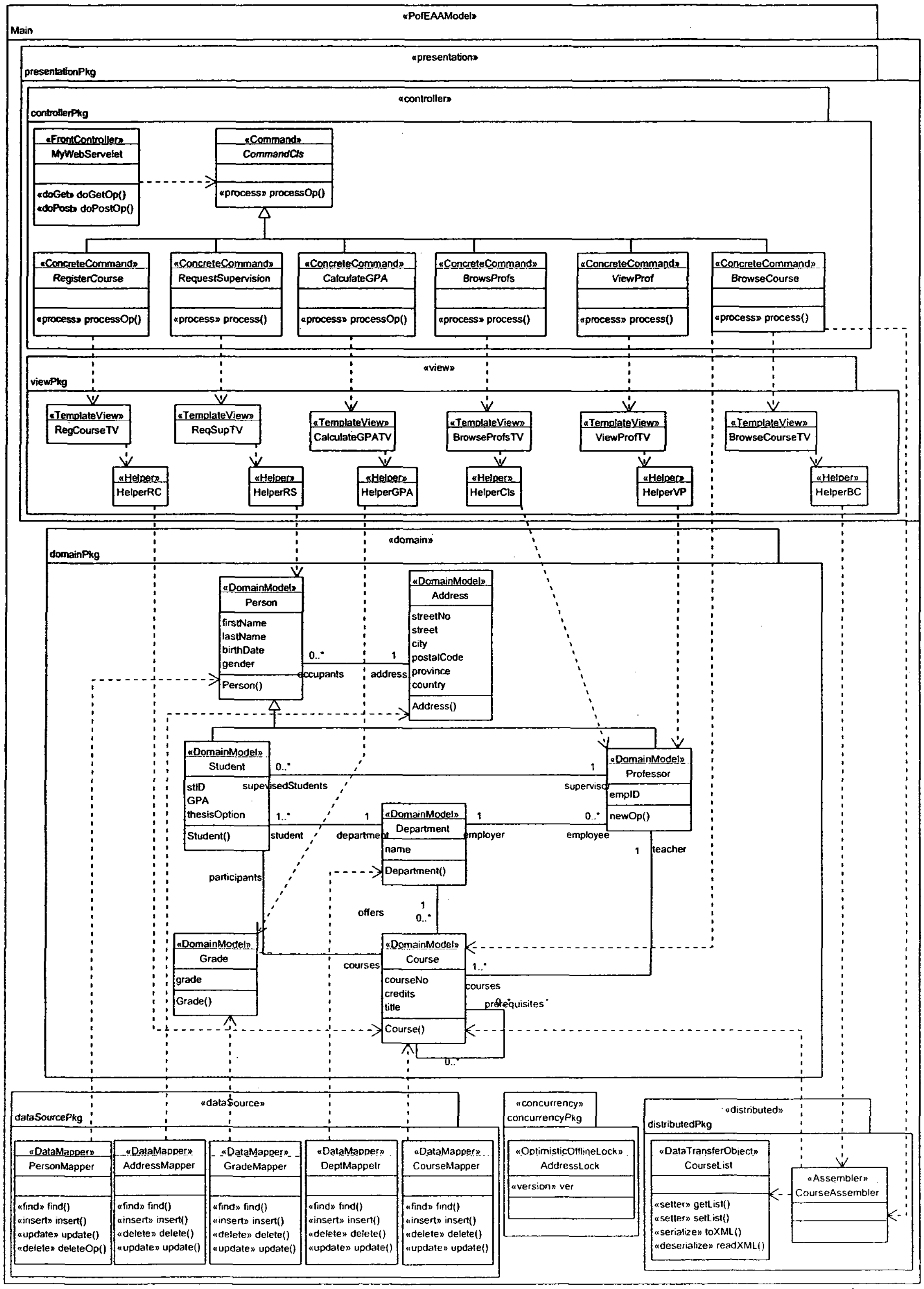


Figure 99: Design of Online Student Registration System - Refined by ArgoPLV

A.7 Design Rationale

Table 23 shows an excerpt of the Design Rationale file which is created during the verification of the design using ArgoPLV.

Table 23: Records from the Design Rationale File Associated with the Repairs

Date/Time	Wizard Class	Issue	Rationale
2009-04-08 11:20:32	WizLayers	PofEAA: Syntactic Problem - Missing Layers in the Model	A design built based upon the PofEAA patterns needs layers such as Presentation, Domain, and Data Source. Other Layers such as Service, Basic, Distributed, Concurrency, and Session State, depend upon the context information set by the tagged values. This wizard has added any of those missing items to the model.
2009-04-08 11:31:45	WizFront Controller	PofEAA: Structural Problem in using Front Controller Pattern	The Front Controller pattern needs a "Handler" class with goGet and doPost operations as well as an Abstract Command class with a Process operation and at least one concrete child. This wizard has added any of those missing items to the model.
2009-04-08 11:39:25	WizViewLayerSem	PofEAA: Semantic Problem regarding the View Layer of the model	The patterns of the View Layer should match with the context information, especially with the value of ViewBuilt tag. This wizard has changed the tag correspondingly.
2009-04-08 16:03:25	WizDomain ModelSyn	PofEAA: Syntactic problem between Domain Model pattern and Data Source Layer	The Domain Layer (from syntactic point of view) should be consistent considering BOTH the relation between patterns in this layer and the Data Source Layer patterns AND and the context information which is set through the TAGGED VALUES. This wizard gives the designer option to change the tagged values correspondingly.
2009-04-08 18:34:27	WizPatterns	PofEAA: Syntactic Problem in organization (layering) of patterns	A design built based upon the PofEAA patterns needs to have each pattern in its corresponding layer.This wizard has rearranged model such that each pattern is placed in the appropriate layer.

Index