Master in Computing
Llenguatges i Sistemes Informàtics

Master's Thesis
– September 2010 –

DESIGN AND IMPLEMENTATION OF A CONCEPTUAL MODELING ASSISTANT (CMA)

Student: *David Aguilera Moncusí*
Advisors: *Antoni Olivé Ramon*
Cristina Gómez Seoane

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Title:	Design and Implementation of a Conceptual Modeling Assistant (CMA)
Author:	David Aguilera Moncusí
Advisors:	Antoni Olivé Ramon Cristina Gómez Seoane
Date:	September 2010

Abstract:

This Master's Thesis defines an architecture for a Conceptual Modeling Assistant (CMA) along with an implementation of a running prototype. Our CMA is a piece of software that runs on top of current modeling tools whose purpose is to collaborate with the conceptual modelers while developing a conceptual schema. The main functions of our CMA are to actively criticize the state of a conceptual schema, to suggest actions to do in order to improve the conceptual schema, and to offer new operations to automatize building a schema.

On the one hand, the presented architecture assumes that the CMA has to be adapted to a modeling tool. Thus, the CMA permits the inclusion of new features, such as the detection of new defects to be criticized and new operations a modeler can execute, in a modeling tool. As a result, all modeling tools to which the CMA is adapted benefit of all these features without further work.

On the other hand, the construction of our prototype involves three steps: the definition of a simple, custom modeling tool; the implementation of the CMA; and the adaptation of the CMA to the custom modeling tool. Furthermore, we also present and implement some examples of new features that can be added to the CMA.

Keywords:	CMA, Design Assistant, Conceptual Modeling, CASE tools, UML
Language:	English
Modality:	Research Work

This thesis was written in LaTeX.
Figures were drawn using Inkscape.

"Make everything as simple as possible, but not simpler."
Albert Einstein

Acknowledgements

First of all, I would like to thank my Master's Thesis advisors Dr. Antoni Olivé and Dr. Cristina Gómez for their support and guidance. Our meetings and discussions made this thesis possible.

I am specially grateful to my colleagues and friends Antonio Villegas, Miquel Camprodon, Rodrigo Pizarro, and Jorge Muñoz. When I was in trouble, their patience and comments were really insightful. I also want to thank Auba Llompart for reading this thesis and correcting my English.

I would also like to thank María for all this time together; she always believed in me and she encouraged me in difficult times.

And, finally, thanks to my family for being always close to me.

This work has been partly supported by the *Ministerio de Ciencia y Tecnología* under TIN2008-00444 project, *Grupo Consolidado.*

This work has been partly supported by the *Universitat Politècnica de Catalunya*, under *Beques UPC Recerca* program.

Contents

List of Figures

1 Introduction

Modeling is hard. The definition of good models is especially hard. In order to reduce the complexity of modeling tasks, and therefore improve schemas' quality, analysts and designers use Computer Aided Software Engineering (CASE) tools. These tools help and assist them throughout all the software development process.

Conceptual modeling, which is in an early stage of the software development process, is the task by which conceptual models are generated. Conceptual models, also known as conceptual schemas, are described in a certain modeling language. A *modeling language* is an artificial language whose purpose is to represent information or knowledge about a domain. A modeling language is defined in terms of its *meta-model*, which states how to define a model, and the rules and constraints its models have to conform to.

Modeling CASE tools use meta-models to check whether a specific model is valid or not. Having valid models that conform to its meta-model is the first step towards the definition of good models. However, such models may not be right and correct. In the same way that a syntactically correct sentence does not ensure that it is also semantically correct, a model that conforms to its meta-model may be incorrect. There are some typical properties that can be automatically tested to determine this other kind of correctness [41], but they are not included in current modeling CASE tools. The aim of this thesis is to address this issue by increasing the capabilities these tools offer.

This chapter begins with a description of what conceptual modeling is and its relevance in the software development process. We define the role modeling languages play in conceptual modeling, focusing our attention in the Unified Modeling Language (UML) and its meta-model. Then, we justify the importance of using modeling CASE tools. We see that they are supposed to provide some sort of help and assistance to analysts and designers in early phases of the development process. However, as we describe in the aim of this thesis, modeling CASE tools are not powerful enough; there is full of requests on what a modeling CASE tool should offer and is currently missing. After briefly analyzing these ideas, we present the goal of this thesis in detail. Finally, we present the outline of the remaining chapters of the thesis.

1.1 Conceptual Modeling

Conceptual modeling is an early activity of the software development process, closely related to requirements engineering. It tries to gather, organize, and classify the relevant, *general* information of a domain, so that, ultimately, it can maintain *concrete* information [40, 56].

For example, think about a piece of software to manage projects within a company. Such a software may know that an employee works in a project as a programmer, and in another project as an analyst, once it knows that there actually are employees in the domain and that an employee may be assigned to a project playing a certain role.

Surprisingly, it is quite common among developers to begin software development without an available conceptual schema. They tend to invest their time in programming rather than designing a conceptual schema. As a result, conceptual schemas are sometimes considered documentation items with little or no value to the resulting software. However, even if there is no specific document with an explicit conceptual schema, someone (the programmer) has to know the general information of the domain; otherwise, it would be completely impossible to code it into a program.

Conceptual schemas are easier to understand than real software due to their higher level of abstraction. As a result, they are much less bound to the underlying implementation technology and much closer to the problem domain [45]. Conceptual schemas provide a piece of documentation that can be discussed and shared with stakeholders and developers. Furthermore, if they are used in a Model-Driven Development (MDD) framework, they can be used to generate code automatically [45].

Conceptual schemas, among other things, include the *structural schema* and the *behavioural schema*. The former consists of the set of entity and relationship types. It is usually known as the static component of the general knowledge. The latter, on the other hand, represents the valid changes in the domain state, as well as the actions the system can perform [40]. In other words, it defines and constraints how the population of the model can evolve.

1.1.1 UML: A Modeling Language

A *modeling language* is an artificial language whose purpose is to represent information or knowledge about a domain. To put it simply, it is "what we use to specify a conceptual schema". These languages, which can be graphical or textual, express the concepts we find in a domain, the relationships between these concepts, some constraints to be satisfied, etc.

There are several modeling languages, but in this master's thesis we only focus on the Unified Modeling Language (UML). UML is a standardized, graphical, general-purpose modeling language maintained by the Object Management Group (OMG). The specification of its latest version (2.2) can be found in [39].

UML covers a large and diverse set of application domains. Not all of its modeling capabilities are required in all domains. Consequently, the language is structured modularly to allow the selection of only those parts that are of direct interest for a given domain. In this thesis, we only use a subset of the UML that permits us to define structural schemas.

In an structural schema expressed in UML we may find concepts (named *Classes*, represented by boxes), their relationships (named *Associations*, drawn using lines that connect the classes), *Generalizations* of a set of classes, etc.

Fig. 1.1 shows an example of how to model a domain using UML. In this example, we can easily see that there are *Employees*, who can either be a *Boss* or a *Regular Employee*. A *Regular Employee* may be assigned to one, two, or three *Projects*, and a *Project* may have as many *Regular Employees* as needed (even none). When a *Regular Employee* is assigned to a *Project*, it plays a certain *Role*. Such *Role* is, in fact, associated to the *Membership* of that *Regular Employee* to that *Project*.

Using this general knowledge, we would now be able to say, for example, that "John", who is a *Regular Employee*, is currently working in the "Eclipse Project" and "VISIO". In the first project he is an "analyst" and in the second one he is a "programmer". This concrete knowledge

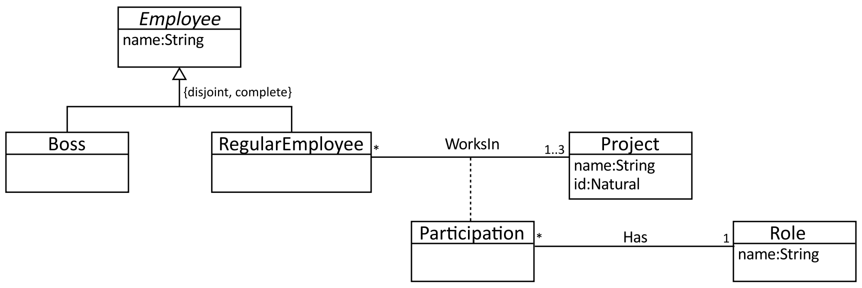


Figure 1.1: Example of a domain modeled in UML.

is an *instantiation* of the general knowledge. Fig. 1.2 shows how to represent this information in UML. This information is usually known as “an instantiation of the model”.

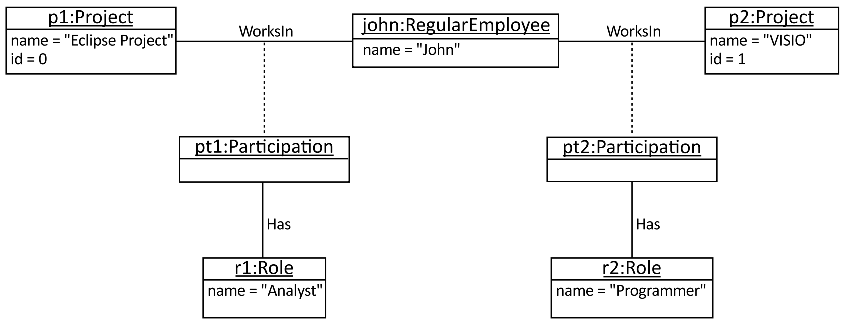


Figure 1.2: Example of an instantiation in UML.

1.1.2 UML's meta-model

A *meta-model* is a precise definition of the constructs and rules needed for creating models [16]. Meta-models can be used as a schema for semantic data that needs to be exchanged and stored, as a language that supports a particular methodology or process, or as a language that expresses additional semantics of existing information.

If we look at Fig. 1.1 we see a domain modeled using UML. The schema shows concepts like *Boss* or *Role*. If we take a closer look, we see that UML uses its own meta-concepts to express domain concepts. For example, both *Boss* and *Role* are *Classes*, and relationships between *Classes* are modeled using *Associations*.

Fig. 1.3 shows a simplified version of UML's metamodel. We can see that any element in a UML schema is, obviously, an *Element*. Some of these elements have an associated name, like *Class* or *Association*; this is why they indirectly are *Named Elements*. A *Generalization*, for example, is a special kind of *Element* that relates two instances of *Classifier*. Note how the constraints defined in the meta-model affect the model: for example, an *Association* has to be related to, at least, two *Properties*.

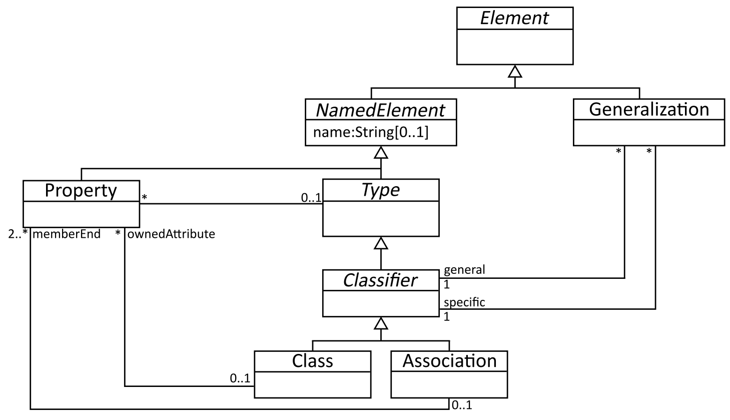


Figure 1.3: A simplified version of the UML metamodel.

1.2 Modeling CASE Tools

The acronym CASE is generally used to refer to "Computer-Aided Software Engineering" [22]. CASE tools are applications that automate, to some extent, the design process of a software product, providing a set of functionalities that help and assist analysts and designers in their daily job.

Modeling CASE tools are focused on the early stages of the software development process, where the time and effort required to locate and debug software problems is much greater. In Fig. 1.4, extracted from [50], we can see that up to 85% of software bugs are due to inaccurate analysis and design specifications.

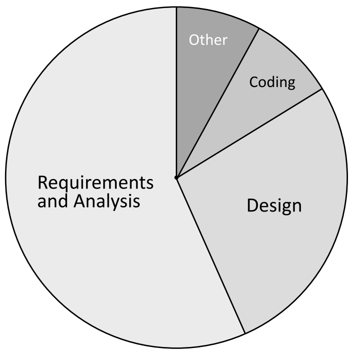


Figure 1.4: Source of software bugs.

CASE tools should dramatically improve analysts' productivity in two areas: initial application system development and long-term maintenance of the production software [50]. The goal is to use this kind of tools to analyze and design an application, while the source code is automatically generated based on the design.

Nowadays, modeling tools offer some sort of support [52]: *consistency checking*, which ensures that the products of the analysis conform to the rules of structured specifications; the usage of a *concrete methodology* to manage the extreme complexity that system development task involves; automatic *code generation* from the specification; etc.

1.3 Aim of this Thesis

As described in [28, 31], current CASE tools are centered in methodology, paradigms and techniques, rather than in users. As a consequence, these tools are too restrictive. A fully assisted modeling environment is a widely unexplored field. We address this issue by designing a Conceptual Modeling Assistant (CMA) which may run on top of current CASE tools to automatically *assist* and actively *criticize* the work modelers do.

A next generation CASE tool should include the following features:

1. It should differentiate between a novice user and a professional one.
2. The functions shown to the modeler should depend on the context.
3. It should guide the user in the development process, trying to figure out what the next step is.
4. It should tolerate inconsistencies, because they are part of the creative process.
5. It should avoid reinventing the wheel; that is, it should provide a wide range of predefined, application domain-specific templates for reuse.

In [22], Gane sketches a few more ideas on the future of CASE. While some of them have already been achieved, a lot of these features remain underdeveloped.

6. It should include expert systems, such as natural-language parsers, that come close to the best human performance at specific tasks.
7. There should be no difference between the development of the new software and the maintenance of the existing one.
8. It should provide real-time feedback on the syntax of model diagrams.
9. It should suggest entity types according to its description, probably using ontologies.
10. It should allow multi-user systems, where large models can be shared by more analysts.
11. It should incorporate a common repository from which to retrieve company's own knowledge.

As we shall see in Ch. 3, several of these ideas have been already implemented into CASE tools. The problem we find in all the features presented in the state of the art is that they are scattered among too many different tools, instead of being under the same platform. Therefore, our aim is to design a piece of software that can integrate as many of these features as possible. These new features are either a new functionality or a criticism to the model:

New functionalities that offer shortcuts to and, thus, automate common tasks (5, 6 and 9). A few examples are applying design patterns, providing a wide range of predefined, application domain-specific templates for reuse, or having a repository from which to retrieve knowledge (11).

Criticisms to modeler's work that outline errors are outlined as soon as they appear (1, 2 and 8). Those errors may be severe, like invalid syntax or schema unsatisfiability, or just recommendations, such as not following a naming guideline.

We believe that the adoption of a tool like the CMA would provide great benefits to its users, but as Hall and Khan explain in [26], "the adoption of new technologies is not always easy". The costs it implies, especially those of the non-pecuniary "learning" type, are incurred at the time of adoption, and cannot be easily recovered. Thus, the CMA should not be a tool itself, but something that can be plugged into current CASE tools. Thus, we would not change how people work, but we would simply broaden the range of available options.

In Ch. 3 we present ArgoUML, a tool that focuses on providing *cognitive support*, that is, criticizing models. Despite ArgoUML's goals are close to our CMA's, the former has some open issues that may prevent modelers from using it. Our CMA addresses these issues and becomes an extensible tool which is able to implement a wide range of the previously enumerated ideas, including the ones supported by ArgoUML, and it is presented as an extension of currently existing CASE tools, so its adoption can be done in a seamlessly manner.

1.4 Outline of the Document

The master thesis is organized as follows:

In Ch. 2, we explain the research methodology used to develop our work, and how it fits to this master thesis.

Chapter 3 reviews the state of the art from different perspectives. First, we analyze current modeling tools in order to know which features, if any, they include to guide modelers. Second, we study a set of tools that implement features aimed to improve the model quality. These features are categorized in the following groups: *understandability improvements*, *schema property checkings*, *inconsistency management*, and *refactorings*. Finally, we see how Integrated Development Environments (IDE) implement many functionalities aimed to *improve* the code quality and *simplify* programmer's work, because it may shed some light on how to help modelers when modeling.

In Ch. 4 we define the goals our CMA has to fulfill, and we describe all the concepts required to understand subsequent chapters. In particular, we explain what a *Platform Tool* is and what its parts are. We then briefly describe how the CMA should be adapted onto it.

Chapter 5 describes our CMA's architecture. First, we present an overview of this architecture. This overview includes the definition of some important concepts related to the architecture and the organization of the architecture in two levels: the *knowledge* and the *operational levels*. Next, we describe in detail each level.

Chapter 6 covers the implementation of the previous ideas. The construction of the prototype includes the design and implementation of a simple modeling tool, the implementation of the CMA itself, and the adaptation of the CMA to the modeling tool.

The CMA prototype we implemented is an extensible tool. New features can be included by implementing a plugin. Chapter 7 presents some plugins we implemented to test our CMA. We have organized them according to their scope and, for each one, we describe it, provide some notes on how it is implemented, and show a few results of its execution.

Chapter 8 concludes the thesis with a short review of the conclusions extracted from this research. It also sketches some notes about future work that has to be done in order to have a fully functional CMA running on top of a real modeling CASE tool.

2 Research Methodology

In this chapter we describe the research methodology used to develop our work. We first define what Design Research is, and we then focus on how the methodology applies in this master thesis.

2.1 Design Research Overview

The work presented here is structured following the main ideas of the Design Research methodology. As stated in [51], Design Research "involves the analysis of the use and performance of designed artifacts to understand, explain, and very frequently improve on" those artifacts.

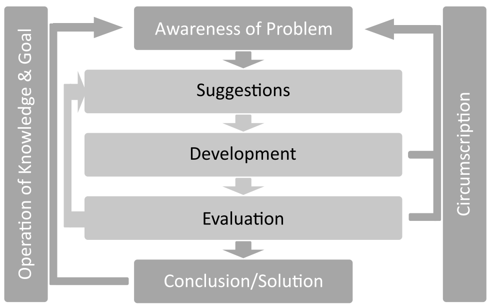


Figure 2.1: Reasoning on Design Cycle.

Fig. 2.1 illustrates the course of a general design cycle, as Takeda et al. analyzed in [48].

This model always begins with *Awareness of a problem. Suggestions* to solve the problem are abductively drawn, using the existing knowledge available. Then, an artifact that implements the proposed solution (*Development* stage) is built. Implemented solutions are then *Evaluated. Suggestion*, *Development* and *Evaluation* are frequently performed iteratively, so better and more accurate solutions can be found. *Conclusion* indicates the termination of a project.

New knowledge production is shown in the figure by arrows labeled *Circumscription* and *Operation of Knowledge and Goal.* The *Circumscription* process is really relevant in design research, because it outlines the *importance of construction to gain understanding.*

Table 2.1 summarizes the outputs that can be obtained from a design research effort [51].

Output	**Description**
Constructs	The conceptual vocabulary of a domain
Models	A set of propositions or statements expressing relationships between constructs
Methods	A set of steps used to perform a task
Instantiations	The operationalization of constructs, models and methods
Better theories	Artifact construction as analogous to experimental natural science

Table 2.1: The Outputs of Design Research

2.2 Design Research in this Master Thesis

The first stage in Design Research is the *awareness of the problem*. In our case, we found that there is a lack of a real conceptual modeling assistant. As stated in Sec. 1.3, nowadays modeling CASE tools provide some level of automatism to the development process, but they do not assist the modeler at all.

In Ch. 3 we review some *research contributions* related to modeling tools and providing assistance. On the one hand, we see the functionalities current modeling tools have, and how they are presented to modelers. This stage is specially important because our goal is to define a conceptual modeling assistant that runs on top of these tools. On the other hand, we study different approaches on providing assistance and, more specifically, how this assistance is implemented in a tool. These functionalities, not included in current modeling tools, are scattered in different applications and have to be integrated, as long as possible, by the CMA.

In chapters 4 to 7 we show the result of iterating over the *Suggestion – Development – Evaluation* cycle. Throught these chapters we propose an architecture that (1) can be adapted to current modeling tools and (2) extended with new functionalities. We study the feasability of this architecture by building a proptotype of the CMA. This prototype is capable of extending the functionalities of an existing modeling tool with plugins that, in fact, extend the power of the CMA itself.

The *Conclusion/Solution* of this research is the architecture of a CMA, along with a prototype that provides some clues on how to adapt the CMA to a real CASE tool.

Fig 2.2 shows graphically how design research was applied to this master thesis:

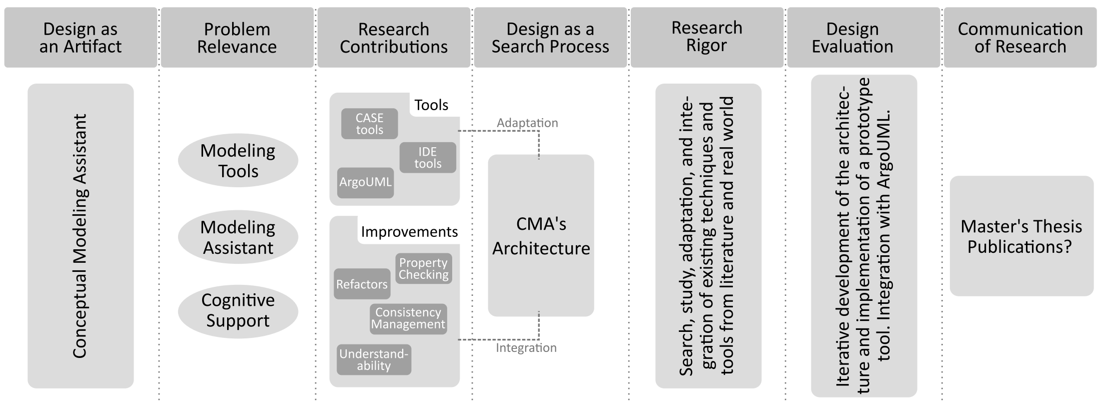

Design as an Artifact
Problem Relevance
Research Contributions
Design as a Search Process
Research Rigor
Design Evaluation
Communication of Research
Conceptual Modeling Assistant
Modeling Tools
Modeling Assistant
Cognitive Support
Tools
CASE tools
IDE tools
ArgoUML
Improvements
Property Checking
Refactors
Consistency Management
Understand-ability
Adaptation
Integration
CMA's Architecture
Search, study, adaptation, and integration of existing techniques and tools from literature and real world
Iterative development of the architecture and implementation of a prototype tool. Integration with ArgoUML.
Master's Thesis Publications?

Figure 2.2: Design Research in this Master Thesis.

3 State of the Art

The aim of this thesis is to design a *tool* that runs on top of *existing modeling tools* and provides a *better assistance* to modelers while they specify conceptual schemas. Hence, this chapter explores the literature related to providing assistance in conceptual modeling and, more specifically, how this assistance was implemented in a tool.

First, we analyze current modeling tools in order to know which features, if any, they include to guide modelers in their job. As we shall see, they all provide some sort of automatisms to improve software development, specially the transition from models to code, but not the modeling task itself. After a brief review of their characteristics, we describe in more detail a couple of tools that, in our opinion, are the most relevant: ArgoUML and MetaEdit+. These two tools are specially interesting because they accomplish some of the goals our CMA has to fulfill.

Second, we study a set of tools that implement features which may dramatically improve the model quality if they are used while developing these models. These features are categorized as follows: *understandability improvements* that simplify the comprehension of the conceptual schemas by end users and modelers; *schema properties checkings*, like ensuring the population of a class is not always empty; *inconsistency management* to ensure a model conforms to its meta-model; and *refactorings* that improve the resulting schema by following a set of guidelines on how to define a good schema.

Finally, we see how Integrated Development Environments (IDE) implement many functionalities aimed to improve the code quality and simplify programmer's work, like *code refactoring* or *code completion.* We believe that the ideas IDEs implement to help their users may shed some light on how to help modelers. We focus our attention in one particular IDE: the Eclipse Platform. We have chosen Eclipse because, although virtually all IDEs provide the same functionalities, it has some interesting plugins whose goal is to improve its assistance, making it specially interesting for our CMA.

3.1 Modeling Tools

In this section, we analyze current modeling tools in order to know the features they provide. After a brief comparison of their characteristics, we describe in more detail ArgoUML and MetaEdit+. These two tools are specially interesting because they accomplish some of the goals our CMA has to fulfill.

3.1.1 Comparison of Modeling Tools

ArgoUML [1] is one of the most complete tools available nowadays. Its latest release, which is 0.30.1 by June 2010, includes plenty of new features:

- UML 1.4 support.
- XMI support.
- Code generation and reverse engineering.
- Design critics, corrective automations, to-do list, check lists, etc.

Eclipse Platform is an extensible IDE. With the Eclipse Modeling Framework (EMF), which includes the UML2Tools plugin [14], Eclipse has a Graphical Modeling Framework editor for manipulating UML models.

- UML 2.1 support.
- MDA support.
- Code generation and reverse engineering.
- Requirements management support.
- XMI support.

Ideogramic UML [30] is a tool for creating UML diagrams. As stated in its website, its main strength is its user interface which, "unlike heavyweight CASE tools with bloated, hard-to-learn interfaces", offers "just the features that you need". The tool focuses on how diagrams are drawn, and provides a completely different approach: *freehand drawing*.

Magic Draw [36]

- UML 2.3 support.
- Traceability from requirements to implementation and deployment models.
- Multi-user environment.
- Code generation and reverse engineering.
- UML Profiles and custom diagrams to extend standard UML.

MetaEdit+ [33]

- Definition of custom modeling languages.
- Multiple views (graphical diagrams, matrices, tables, etc.)
- Multi-user environment.
- Code generation.

Poseidon for UML [24]

- UML 2.0 support.
- Template-based code generation for different programming languages and reverse engineering for Java.
- UMLdoc documentation generation.
- XMI support.

Rational Rose Modeler [29]

- UML 1.x support.
- MDD support through Patterns.
- Team support through a merge mechanism.
- Web publishing and report generation.

USE [25] can parse and interpretate OCL expressions in order to validate the correct specification of the system. It also allows the instantiation of the model so the analyst can check whether the constraints hold or not.
USE lacks a user interface to define a UML schema. Therefore, it has to be defined textually before using the program. OCL expressions can be defined at run-time, so the user can check certain properties once the schema has been loaded. However, those expressions are not stored in the resulting schema; they have to be typed manually using an external text editor.

Visual Paradigm [55]

- Requirements management.
- Impact analysis support.
- Multi-user environment.
- Ming mapping (brainstorming tool).
- Report generation.
- XMI support.

As we can see from the previous features list, the general operation of every single tool is almost the same. To our understanding, almost all these tools are tightly related to the coding stage, which means that the emphasis is given in the link between models and code by means of code generation and reverse engineering. From a modeling point of view, the features they offer, such as multi-user environments or definition of patterns, are insufficient. The only tool that really focuses on the modeling stage and, thus, tries to improve models' quality is *ArgoUML*.

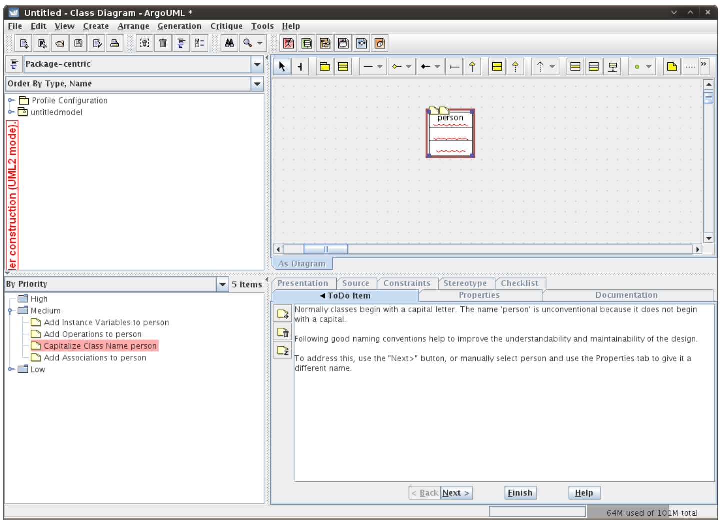
Note that *USE* and *Ideogramic UML* are also exceptions, because of the goals they pursue. The former is not a modeling tool at all, because it focuses on checking a schema, instead of defining it. The latter tries to simplify the design process at the expense of reducing the functions the environment offers. It focuses on drawing the schema and adding annotations, among other functions, which may be interesting for non-expert users, but a handicap for more expert ones.

3.1.2 ArgoUML

As stated in [42], ArgoUML is a *domain-oriented* design environment that provides *cognitive support* of object-oriented design. It provides some of the same automation features of a commercial CASE tool, but it focuses on features that support the cognitive needs of designers. Fig. 3.1 shows the ArgoUML's User Interface criticizing modeler's work.

ArgoUML is particularly inspired by three theories within cognitive psychology [42]: (i) reflection-in-action, (ii) opportunistic design and (iii) comprehension and problem solving.

Reflection-in-action This theory observes that modelers do not conceive a fully-formed design. Instead, they construct a partial design and evaluate it so that, ultimately, they can revise, improve and extend it.



Normally classes begin with a capital letter. The name 'person' is unconventional because it does not begin with a capital.

Following good naming conventions help to improve the understandability and maintainability of the design.

To address this, use the "Next>" button, or manually select person and use the Properties tab to give it a different name.

Figure 3.1: ArgoUML's User Interface showing some improvements available.

Opportunistic design A theory which states that, despite the fatct that users plan and describe their work in an ordered fashion, in the end they choose successive tasks based on the criteria of cognitive cost.

Comprehension and Problem Solving The theory notes that designers have to bridge a gap between their mental model of the problem or situation and the formal model of a solution or system.

ArgoUML implements these theories using a number of techniques:

- A user interface which allows the user to view the design from a number of different perspectives.
- Processes running in parallel with the design tool that evaluate the current design against models of what "best practice" design might be like (*design critics*).
- The use of *to-do lists*, so the user can record areas for future work.
- The use of checklists, to guide the user through a complex process.

Despite this new approach makes ArgoUML really useful at the modeling stage, the tool lacks some features that are very important and common among other tools. The absence of an *undo/redo* mechanism and the *copy/paste* meta-commands may discourage modelers from using ArgoUML. In fact, and as stated in [38, sec.3], an *undo/redo* mechanism has been one of the most requested features for ArgoUML. Although its implementation work has begun, it is not yet at a usable state, due to the complexity associated with the handling of model element deletion. There is an open issue pointing this problem out since 2003 in [49].

Design Critics

The key feature that distinguishes ArgoUML from other UML CASE tools is its use of concepts from cognitive psychology. The *critics* are background processes which evaluate the current model according to various "good" design criteria. There is one critic for every design criterion.

The output of a critic is a *critique*, which points out some aspect of the model that does not appear to follow good design practice. A critique generally suggests how the bad design issue can be rectified.

ArgoUML categorizes critics according the design issue they address[1]:

- Uncategorized
- Class Selection
- Naming
- Storage
- Planned Extensions
- State Machines
- Design Patterns
- Relationships
- Instantiation
- Modularity
- Expected Usage
- Methods
- Code Generation
- Stereotypes
- Inheritance
- Containment

A couple of examples of these critics are the detection of duplicate class names, which violates the condition that “names of contained elements in a namespace have to be unique”, and the suggestion of concrete subclasses definition when a class is abstract, because such a class “can never influence the running system because it can never have any instances”.

Currently, the implementation of critics detection is quite simple. There is a background process that evaluates the current model periodically against all the critics implemented. There is a *critiquingInterval* variable that determines how often the critiquing thread executes. As the ArgoUML programmers say, the concept of an interval between runs will become less important as ArgoUML is redesigned to be more trigger driven.

3.1.3 MetaEdit+

MetaEdit+, proposed by Kelly, Lyytinen and Rossi in [33], is a CASE tool that resolves, to a greater or lesser extent, some of the issues listed in Sec. 1.3. It “enables flexible creation, maintenance, manipulation, retrieval, and representation of design information among multiple developers”.

One of its main characteristics is its multi-user nature. MetaEdit+ can be run either as a single-user workstation, or simultaneously on many workstation clients connected to a server.

The heart of its architecture is the MetaEngine, illustrated in Fig. 3.2, which handles all operations on the underlying conceptual data. The MetaEngine provides a common way to access the repository data, where design objects are stored. Every new single functionality added to the system runs on top of this MetaEngine, so each one is only responsible of itself, without interfering the others.

In the design of MetaEdit+, tools have been classified into five distinct families:

Environment management tools are used to manage features of the environment, its main components, and to launch it.

Model editing tools are used to create, modify and delete model instances, and to view them from different view points.

[1]See [42, Ch. 15] in order to view a detailed description of each category and the critics within it.

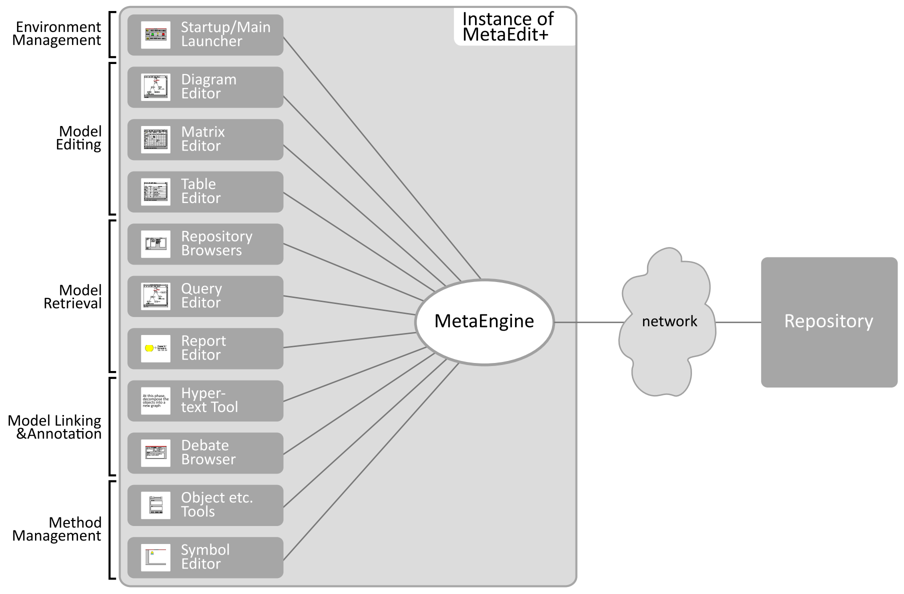


Figure 3.2: Architecture of MetaEdit+.

Model retrieval tools are used for retrieving design objects from the repository (regardless of the fact that they are models or metamodels).

Model linking and annotation tools are used for traceability, or maintaining conversations about design issues, etc.

Method management tools are for method specification, management and retrieval.

3.2 Model Improvements

In this section, we study a set of tools that implement features which may dramatically improve models' quality if they are used while developing these models. These features are categorized as follows:

Understandability Improvements of conceptual schemas, both for regular stakeholders and highly-skilled users.

Checking Properties of the Schemas , like ensuring the population of a class is not always empty (*satisfiability*).

Inconsistency Management , which involves the detection, handling, and eventually reparation, or inconsistencies that arise while creating the models.

Refactorings to improve the final model by including best-practices to solve common problems.

3.2.1 Improving the Understandability of Conceptual Schemas

Conceptual design is a very important aspect of information systems development. Wrong conceptual models can lead to serious problems since the software trusts that the right concepts and relations are chosen. On the one hand, much effort has to be spent on the communication and negotiation with the stakeholders, which means that they have to understand conceptual schemas. On the other hand, the largeness of conceptual schemas makes it difficult for users to get the knowledge of their interest, regardless they are regular stakeholders or highly-skilled users [34, 54]. In this section, we present different tools to improve the understandability of conceptual schemas: *paraphrasing tools* that generate a natural language description of the schema, and a *filtering tool* that reduces the complexity of a schema by focusing on a few classes.

As Kop states in [34], the most used representation of conceptual modeling is a graphical representation, which is good for IT professionals, but end users are typically not able to understand them. Therefore, it is required to verbalize the conceptual schema, that is, to transform it back to natural language.

In [37], Meziane, Athanasakis and Ananiadou present the GeNLangUML system, which generates English specifications from UML class diagrams.

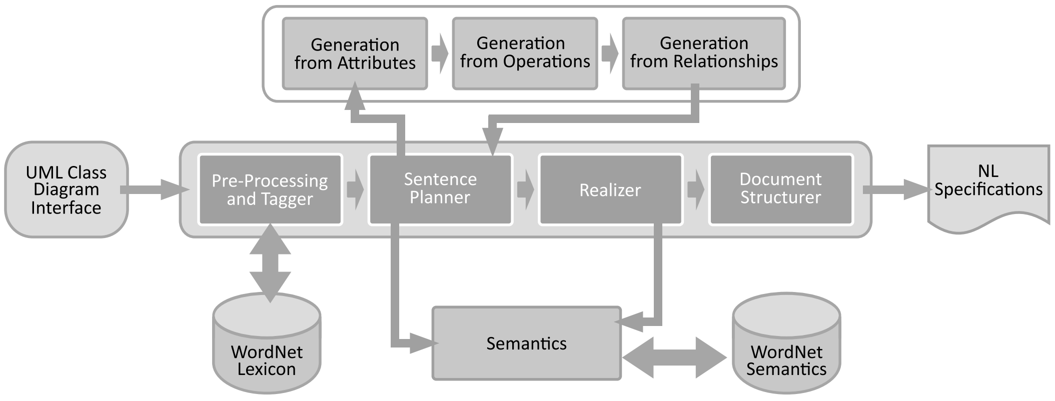

Generation from Attributes
Generation from Operations
Generation from Relationships
UML Class Diagram Interface
Pre-Processing and Tagger
Sentence Planner
Realizer
Document Structurer
NL Specifications
WordNet Lexicon
Semantics
WordNet Semantics

Figure 3.3: Architecture of the GeNLangUML system.

Fig. 3.3 illustrates the architecture of the GeNLangUML system, which is very similar to that found in generic Natural Language Generation (NLG) Systems. It is composed of four components: the *pre-processor and tagger*, the *sentence planner*, the *realizer* and the *document structurer*:

- The *UML Class Diagram Interface* was designed to define UML models. The internal representation they use is based on two XML files: the former stores the classes and the latter the relationships.
- The *pre-processing and tagging* processes the input and tries to disambiguate it. It uses a controlled language, because controlled languages result into fewer ambiguities, as the input's variation is limited.
- The *sentence planner* component receives the tags and the corresponding words from the tagger and generates sentences following templates. Each template defines how to verbalize attributes, operations and relationships.
- The *realizer* ensures agreement between words and aggregations.
- The *document structurer* structures the generated sentences into an output in a more readable format. Sentences referring to the same entities are generated at the same time, so there is a flow in the generated text.

Halpin and Curland propose in [27] an extension to NORMA (an open-source software tool that facilitates entry, validation, and mapping of ORM 2 models) to verbalize these models. They wanted to meet five main design criteria: expressibility, clarity, flexibility, localizability, and formality.

The verbalization support implemented uses a pattern-driven generative approach whenever possible. The rules for verbalization of a constraint pattern are constant, but the actual text used depends on environment-specific factors, such as the language or the output format.

The implementation, which uses *field replacement* to simplify the verbalization engine and to enable data-driven snippet sets to be specified according to both language and user preferences, was broken down into the following components: the *selection manager* and the *snipped manager*.

In [32], Kalyanpur et al. present an algorithm that provides Natural Language (NL) *paraphrases* for OWL Ontologies on the Semantic Web. They describe the design and implementation details of their NL algorithm in the context of an existing semantic web ontology engineering toolkit: SWOOP.

Fig. 3.4 illustrates the architecture of SWOOP, along with the additional plugin implemented to generate a natural language representation of OWL concepts. As shown in the figure, various kinds of renderer can be plugged in SWOOP. Their work was to design a Natural Language Entity Renderer plugin.

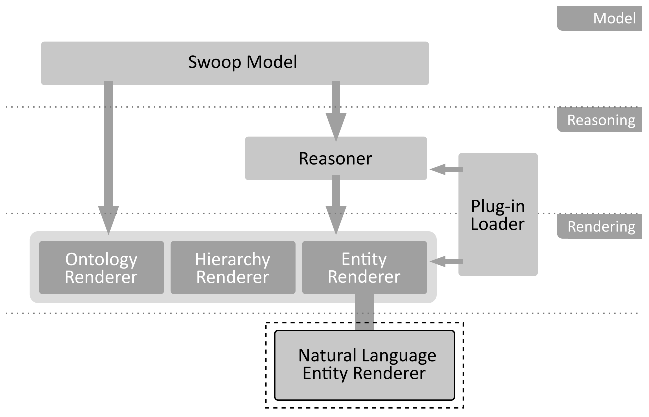


Figure 3.4: The Natural Language Entity Renderer Plugged in SWOOP.

In their work, they define how to parse the taxonomy using the *Visitor Design Pattern* and, hence, to generate a natural language parse tree. Once they have this tree, the task of displaying a NL output of the OWL class expression reduces to walking this tree and printing out the values of nodes and links in an orderly fashion. Additionally, to improve results, they use a combination of filters and rules to sort the links, aggregate relevant information and combine data about different restrictions on the same property.

A different problem related to the understandability of a conceptual schema arises when we are dealing with large conceptual schemas. As Villegas and Olivé expose in [53, 54], the largeness of a conceptual schema makes it difficult for users to understand it and to extract knowledge from it. Thus, they argue that computer support is mandatory, and they propose a new method, which they implement into a prototype tool, to improve the usability of these schemas.

In the former paper, they present a method to compute the *importance* of every entity type in a schema. In order to compute this metric, they implemented different algorithms found in the literature, and also adapted them to include additional information, such as textual integrity constraints. Once the importance is computed, the top n entity types are returned.

In the latter paper, the approach is quite different. Instead of returning the most important classes in the schema, they return the most *interesting* classes according to what the user is interested in. Basically, they query the user which classes she wants to know more about, and the method returns a filtered set with the most interesting classes related to those. In order to decide the most *interesting* classes, they compute the *closeness* and *importance* factors from the structure of the original conceptual schema.

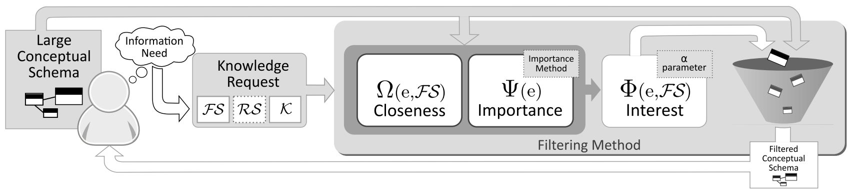


Figure 3.5: Overview of the Filtering Method proposed by Villegas and Olivé in [54].

Fig. 3.5 illustrates their filtering method. The user is interested in retrieving more information about a portion of a schema, but it is too large to do it manually. The method requires the user to introduce some data that describes what she wants; that is, a *filtering set* $\mathcal{FS}$, which defines which entity types is the user interested in and wants to know more information about, a *rejection set* $\mathcal{RS}$, which eventually defines those entity types the user does not want, and how many $\mathcal{K}$ classes the result has to contain.

In order to prove the effectiveness of the method they proposed, the authors implemented a prototype tool on top of the USE environment [25] and tested it.

3.2.2 Checking Properties of the Schemas

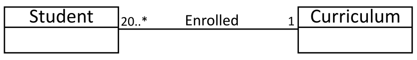
The correctness of a conceptual schema can be determined by reasoning on the definition of the schema itself. There are some typical properties that can be automatically tested, like schema satisfiability or operation executability [41]. In this section, we review different tools that test the satisfiability of a schema, that is, whether a class is forced to have either zero or infinitely many objects.

Consider the example presented in [9] by Cadoli et al., and illustrated in Fig. 3.6. The example refers to an application concerning management of administrative data of a university, and exhibits two classes and an association between them. The multiplicity constraints state that:

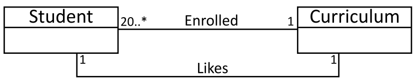
- Each student has to be enrolled in one, and only one, curriculum.
- Each curriculum has to have, at least, twenty students.

Note that, because of these constraints, it is impossible to have any number of students between one and nineteen. If we had, e.g., five students enrolled to one curriculum, the constraint stating that "a curriculum has, at least, twenty students" would be violated.

In some cases the number of objects of a class is forced to be zero. For example, consider Fig. 3.6(b), where a new association *likes* has been added. The cardinalities of this association state that a student likes one, and only one, curriculum, and a curriculum is liked by one, and only one, student. Because of these cardinality constraints, the populations of *Student* and



(a) A UML class diagram.



(b) A UML class diagram with finitely inconsistent classes.

Figure 3.6: Simple UML example of schema satisfiability.

Curriculum have to have the same size, so that the cardinality constraints linked to the *enrolled* association can never be satisfied.

It is obvious that the situation described by Fig. 3.6 is not realistic, but such inconsistencies may arise in more complex situations. As stated in [7], software verification is one of the long-standing goals of software engineering, specially in the context of MDD and MDA, where models are used to (semi-) automatically generate the implementation of the final software. In this section, we will review different approaches to detect these problems.

Cabot, Clarisó and Riera use in [7] Constraint Programming to verify UML/OCL class diagrams. This paradigm provides a fully automated, decidable and expressive verification of these diagrams. A *finite* solution space is established so the constraint solver can perform a *complete search* within this space.

In their paper, the authors describe how to translate a UML/OCL class diagram into a Constraint Satisfaction Problem (CSP). The tool they implemented is base on a set of ECL^{i}PS^e constraint libraries and Java classes, extended with the libraries of the Dresden OCL toolkit and MDR.

In [9], Cadoli et al. also implement *finite model reasoning* on UML class diagrams. They show that it is possible to use off-the-shelf tools for constraint modeling and programming for obtaining a finite model reasoner. Their implementation is based on the standard language Managed Object Format (MOF) to represent UML class diagrams textually, which is then converted to an output language (OPL) that can be used with the state-of-the-art solvers. Thus, their approach is quite similar to the one presented by Cabot et al., but without taking into account OCL constraints.

Queralt and Teniente's main contribution in [41] is an approach to help validating a conceptual schema *with a behavioural part.* It is important to take into account the behavioural part because, although it is possible to find instances of a class satisfying all the constraints, it may be the case that there is no operation that successfully populates it.

To study the feasibility of their approach, they have used the CQC Method [17], which is an existing reasoning procedure, to perform the tests. To check if a certain property holds in a schema, it has to be expressed in terms of an initial goal to attain and the set of integrity constraints to enforce, and then ask the CQC Method to attempt to construct a sample information base to prove that the initial goal is satisfied without violating any integrity constraint.

3.2.3 Inconsistency Management

Another example where modeling CASE tools may be really helpful is consistency checking. Large systems and specifications are rarely consistent. Even if they are, its evolution tends to introduce inconsistencies. Software systems and software specifications are no exception. As Balzer says in [2], exceptions inevitably arise in the data managed by practical software applications. In [46], Spanoudakis and Zisman focus their attention in an earlier stage, and they state that the construction of complex software systems, generally involving various stakeholders, results in many partial models that are not consistent between them.

Due to the increasing use of models, and that inconsistencies inevitably arise during software development, model inconsistency detection is gaining more and more attention. Solutions that detect such inconsistencies as they arise, in a way like IDEs do, lead to interesting design approaches and implementations.

Inconsistency handling involves two main steps: *detection* and *response*. The first one involves detecting the insertion of an inconsistency into the model. The second, involves deciding what to do with the inconsistency found: should it be rejected? Should it be accepted, but marked somehow for later correction? If we accept inconsistencies, should the modeling tool provide a repair plan in order to fix them? There is plenty of literature about these questions, but we only focus on the first stage: detecting inconsistencies.

Blanc et al., in [5], propose an incremental consistency checker based on the idea of representing models as sequences of primitive construction operations. The four elementary operations they define are: *create*, *delete*, *setProperty*, and *setReference*.

In order to detect an inconsistency, they define *Inconsistency Detection Rules*. Any inconsistency rule is a logic formula over the sequence of model construction operations. In other words, if a set of operations were triggered in a specific order, we can assure that an inconsistency has been introduced into the model.

In [15], Egyed proposes an instant consistency checking for the UML. Its immediacy makes it similar to the one proposed by Blanc et al., but it takes a completely different approach to deal with inconsistency detection. His approach defines a few consistency rules for UML 1.3 and checks whether they hold or not each time a change is performed. His main contribution involves the *detection of scope*. The rules have to be checked against the elements that have changed or can be indirectly affected by the change, not against the whole schema. In previous methods, rules were defined in terms of types, but Egyed suggests a new solution where rules are checked against concrete instances, not types.

3.2.4 Refactorings

As introduced in [6, 19], refactorings are changes made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. They describe what can be changed, how the modification can be done without altering the semantics, and what problems to look out for when doing so.

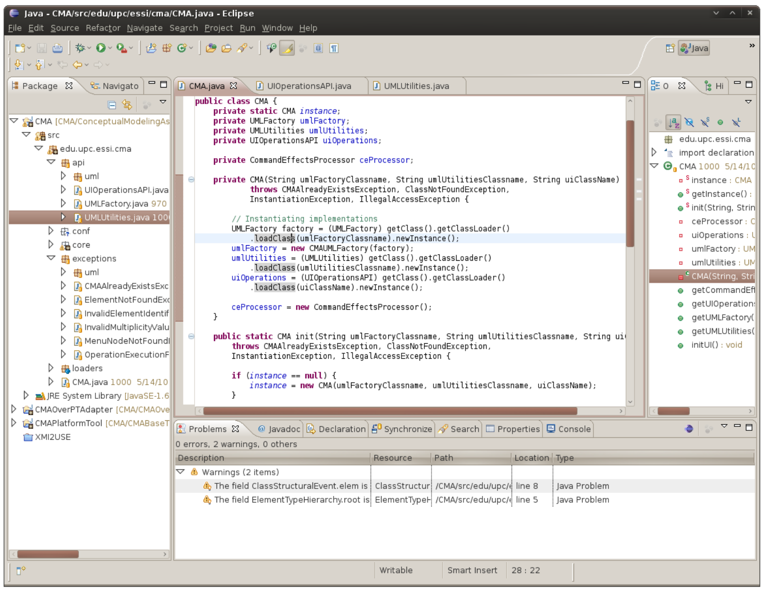
Fowler et al., in [19], argue that automated tools that support refactoring would improve model's quality. Even with the safety net of a test suite in place, refactoring by hand is time consuming. This simple fact prevents programmers from making refactorings they know they should, simply because refactoring costs too much.

Until now, refactorings have usually been discussed in the context of program code. As stated by Boger et al. in [6], refactorings may be defined on the level of models, so refactoring browsers could be implemented in the context of UML CASE tools rather than IDEs. Following this idea, they have implemented a few refactors within Poseidon for UML [24]. Usually, code based refactorings are shown to the user via application menus. The authors, though, have implemented them using panes that can always be seen aside the model. Refactorings are proposed based on the current selection. If, for example, a method is selected, the refactoring *Rename Method* is proposed. Possible conflicts or problems that can occur if the refactoring is executed are also shown.

3.3 The Eclipse Platform: an Example of an IDE

Integrated Development Environments (IDE) implement many functionalities aimed to improve code's quality and simplify programmer's work, like *code refactoring* or *code completion*. We believe that the ideas IDEs implement to help their users may shed some light on how to help modelers. In this section, we focus our attention in one particular IDE: the Eclipse Platform. We have chosen Eclipse because, although virtually all IDEs provide the same functionalities, it has some interesting plugins whose goal is to improve its assistance, making it specially interesting for our work.

As defined in [3], "the Eclipse Platform is designed for building IDEs that can be used to create applications as diverse as web sites, C++ programs, or Enterprise JavaBeans". Fig. 3.7 shows a screenshot of the Eclipse Platform. Eclipse was originally developed by Object Technology International, before IBM bought the company. IBM began working on it to integrate its many development programs [23]. Eclipse was created as an extensible tool, where interoperable plugins can be added to extend its capabilities. Thus, Eclipse can work with a great range of programming languages and applications.


Java - CMA/src/edu/upc/essi/cma/CMA.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package
Navigato
CMA.java
UIOperationsAPI.java
UMLUtilities.java
Writable
Smart Insert
28 : 22

```
public class CMA {
    private static CMA instance;
    private UMLFactory umlFactory;
    private UMLUtilities umlUtilities;
    private UIOperationsAPI uiOperations;

    private CommandEffectsProcessor ceProcessor;

    private CMA(String umlFactoryClassname, String umlUtilitiesClassname, String uiClassName)
            throws CMAAlreadyExistsException, ClassNotFoundException,
            InstantiationException, IllegalAccessException {

        // Instantiating implementations
        UMLFactory factory = (UMLFactory) getClass().getClassLoader()
            .loadClass(umlFactoryClassname).newInstance();
        umlFactory = new CMAUMLFactory(factory);
        umlUtilities = (UMLUtilities) getClass().getClassLoader()
            .loadClass(umlUtilitiesClassname).newInstance();
        uiOperations = (UIOperationsAPI) getClass().getClassLoader()
            .loadClass(uiClassName).newInstance();

        ceProcessor = new CommandEffectsProcessor();
    }

    public static CMA init(String umlFactoryClassname, String umlUtilitiesClassname, String ui
        throws CMAAlreadyExistsException, ClassNotFoundException,
        InstantiationException, IllegalAccessException {

        if (instance == null) {
            instance = new CMA(umlFactoryClassname, umlUtilitiesClassname, uiClassName);
        }
```

Problems | Javadoc | Declaration | Synchronize | Search | Properties | Console

0 errors, 2 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (2 items)				
The field ClassStructuralEvent.elem is	ClassStructur	/CMA/src/edu/upc/e	line 8	Java Problem
The field ElementTypeHierarchy.root is	ElementTypeH	/CMA/src/edu/upc/e	line 5	Java Problem

Figure 3.7: User Interface of the Eclipse Platform.

Fig. 3.8 illustrates the architecture of the Eclipse Platform. When Eclipse is started, it discovers the set of available plugins and builds an in-memory plugin registry; whenever a plugin's functionality is required, the plugin is loaded.

As we briefly introduced, Eclipse can be a modeling tool when used in conjunction with the *Eclipse Modeling Framework (EMF)*. This extension allows an Eclipse user to define models graphically. Despite Eclipse includes a huge set of operations and features that automatize and assist programmers when coding, it does not provide the same or similar functionalities when modeling. Nonetheless, we believe that the following features, which are available at the coding stage, may shed some light on how to assist users:

- Keyword and syntax coloring.
- Compiler problems shown as annotations, outlined as soon as they are introduced.

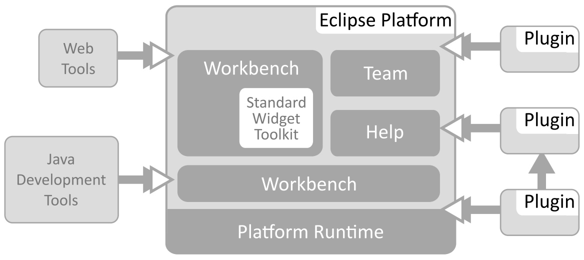


Figure 3.8: Architecture of the Eclipse Platform.

- Code refactoring.
- Code completion.
- Replacement of individual Java elements with versions of element in local history.

3.3.1 Related Work on Improving Eclipse's Assistance

Robbes and Lanza explore in [43] how code completion can be improved. As they state, code completion is one of the top commands executed by developers, along with *copy*, *paste*, *save*, and *delete*. Code completion is "one of those features that once used becomes second nature", and it is integrated in most of the major IDEs.

Their goal is to analyze how this feature is implemented in different systems and to discuss different code completion algorithms. Based on their analysis, they come to the conclusion that, nowadays, it has some limitations. For example, if an API is quite large or the language is not typed, the number of candidates to choose from will still be too large, making the code completion unusable. However, there is little research to improve code completion. They propose the introduction of a new code completion algorithm named "optimist completion", which "performs better than anyone".

In [35], Layman et al. present MimEc, an "Intelligent User Notification of Faults". Its purpose is to display only those faults in which a developer may be interested.

Nowadays, Automatic Fault Detection (AFD) tools show an alert to the programmer as soon as a potential fault has been introduced. The problem is that many of these notifications are produced too often, distracting the programmer. Thus, the programmer's confidence in the tool is reduced and the alerts are generally dismissed, so all the advantages they would provide become useless.

The authors propose a new platform for displaying alert information to the user. This platform is based on the AWARE plugin for Eclipse. AWARE collects information of third-party tools, such as the Eclipse compiler, estimates the potential faults, ranks the faults according to the likelihood that it is not a false positive, and displays the alerts to the user. Their goal is to supplement AWARE with an intelligent interface component, so the alerts presented to the user are both *interesting* and *informative*.

In [13], Dubinsky et al. present an Eclipse plugin to manage User Centered Design (UCD). The UCD approach is used to develop software products acquiring as much feedback as possible, and as soon as possible, from end users. Its goal is to increase the usability for the users by involving them in design and development activities. The authors identified a lack of *UCD management* within the development environment of a project, so they have implemented a tool that fits in Eclipse and solves this problem. The main feature of the UCD management plugin is the ability to create and deploy user experiments within the Eclipse IDE.

3.4 Conclusions

We analyzed the state of the art from two different view points. On the one hand, we presented many current modeling tools and we discussed the features they offer. On the other hand, we explained different approaches to improve the quality of the resulting models. These functionalities include model's understandability improvements, schema properties checks, inconsistencies management, and refactorings. They all have been implemented, somehow, in prototypes, but we think that including them into modeling tools is highly recommendable. Furthermore, we briefly analyzed how assistance is provided in IDEs, because they are applications closely related to modeling tools and the ideas IDEs implement shed some light on how modeling tools can provide assistance.

After the analysis of the literature, we can conclude that the vast majority of modeling tools do not provide real assistance to their users when modeling. They only focus on automating tasks, instead of both automating and criticizing modeler's work. Moreover, the automatisms they offer is not focused on modeling tasks; take code generation as an example. In fact, code generation is of one of current CASE tools' strengths: code is automatically generated from the model, which is really interesting in a Model Driven Development (MDD) environment, but it does not provide any guidance while developing the model itself, even if in an MDD approach the emphasis is given to models.

Yet, there is one exception: ArgoUML. This tool provides cognitive support of object-oriented design. It actively anlayzes and, if required, criticizes the models that are being done. Its aim is to guide modelers while defining conceptual schemas so, ultimately, the resulting schemas are better, follow certain guidelines, and become error-free.

However, despite ArgoUML implements great features, we think that it is not sufficient. There are certain factors that can be improved and, thus, have to be addressed. First of all, other current CASE tools lack this kind of features. We strongly believe that these features should be adopted by them, and it should be done as seamless and easy as possible. The fact that ArgoUML supports critics does not solve that others do not. Furthermore, ArgoUML is still in an alpha (though quite usable) stage. As a result, the way it implements critics may produce some problems in the long-term:

- The detection of critics is done by a background process. This process checks if none of the critics is violated and, if it is, a critique is created. Such an implementation is really inefficient, because these checks are performed regardless of whether the model changed or not.
- ArgoUML has some limitations that may prevent modelers from using it and, therefore, taking advantage from its critics system. For example, it lacks a few commands that are very common among other tools, such as *undo/redo* and *copy/paste.*
- New critics cannot be easily integrated within the environment; they are part of the core. Consequently, whenever a new critic is developed, ArgoUML has to be recompiled and a new version has to be released.

As we have seen, current IDEs provide some guidance to programmers. Keyword and syntax highlighting, code completion, or code refactoring are a few examples of this guidance. These ideas can be ported to a modeling stage within modeling CASE tools. For example, code completion at a higher level of abstraction could imply the use of ontologies, so whenever a new class is created, its attributes are automatically fetched and proposed.

The inclusion of the presented model improvements in modeling tools permit the fulfillment of the ideas listed in Sec. 1.3. The *improvement of the schema understandability* helps novice and expert users to understand the information represented in a model. *Inconsistency management* is a way to tolerate inconsistencies, and along with *checking schema properties*, to improve the resulting model, which is the goal that, ultimately, our CMA pursues. *Refactorings* do also improve the quality of a schema and, as we have seen, provide a mechanism by which information from ontologies can be properly included to our models.

4 Overview of the CMA

In this chapter we present a detailed overview of the CMA, so that subsequent chapters can be easily understood. We first state which goal the CMA pursues and which requirements it is constrained by. We then define what a Platform Tool is and the parts that it has. Finally, we roughly explain how the CMA is supposed to be adapted to a Platform Tool.

4.1 Goals and Requirements

As we have sketched throughout Ch. 1, modeling tools are not powerful enough to offer a complete assistance to modelers. In Sec. 1.3, we have seen several examples on how to offer this assistance to a user. After the literature review from the previous chapter, we came to the conclusion that all the presented solutions are scattered among different tools, instead of being under the same, unified platform.

The goal of this master thesis is to design the architecture of a piece of software that can integrate as many *guidance* and *assistance* features as possible on top of current modeling tools. We have already seen that these features can be *new functionalities* that offer shortcuts to, and thus automate, common tasks, or *criticisms* to modeler's work, so errors are outlined as soon as they appear. We have also stated that we want them to be components that can be plugged in the CMA, in order to extend its capabilities and power.

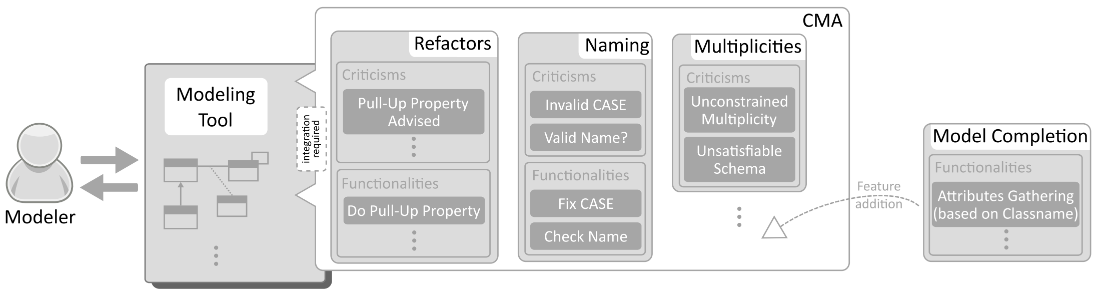


Figure 4.1: The CMA with new features adapted to a modeling tool.

Fig. 4.1 sketches, in a very abstract way, the architecture of the CMA. The CMA is somehow adapted to a modeling tool, and integrates a set of new features that can be plugged in or removed from the CMA. A few concrete examples of the features that could be added to the CMA are also illustrated. Note that a feature may be presented as a combination of one or more functionalities and criticisms:

Naming could provide a set of guidelines that include criticisms to outline invalid names, and a set of operations to validate those names or to fix case.

Refactors could be a set of operations to perform a refactor and criticisms to detect whenever a refactor could be applied.

Multiplicities could include a criticism to point out that a multiplicity constraint has the default unconstrained value, so the modeler notices that it may be changed, or a criticism that identifies whether the schema is satisfiable or not.

Model completion could provides an operation that decides which attributes can be included in a class based on its name. These attributes could be gathered from, for example, a general ontology.

4.2 A Platform Tool

One of the requirements the CMA has to meet is that it has to run on top of existing modeling tools[1], so the former increases the features the latter offers to modelers. The CMA is tightly related to the underlying modeling tool, so it is mandatory to analyze and comprehend how a modeling tool works. In this section, we describe the architecture of a Platform Tool[2] and the assumptions we make about it, so we can then build our CMA.

It is quite reasonable to think that a Platform Tool is designed following a multi-layer architecture, like the one shown in Fig. 4.2. Such an architecture allocates the responsibilities of an application into different layers. Typically, a multi-layer architecture comprises the following layers:

Presentation tier (*User Interface*) The main function of this layer is to translate results to something the user can understand, and to gather information from him to perform new operations.

Logic tier (*Domain — UML Model*) This layer processes commands, performs calculations, and moves information between the two surrounding layers. Usually, this layer loads the information from the Data tier and performs the operations without interacting with the Data tier, unless the user wants the changes to be saved.

Data tier (*XMI Reader/Writer*) The information is stored and retrieved from a database or file system.

Usually, the user interface of a Platform Tool is a Graphical User Interface (GUI), which means that the operations the modeler can perform and the information of the model are shown graphically. Some of the most common GUI elements, shown in Fig. 4.3, are:

- *Menus* and *tool bar buttons* to present the available operations.
- *Shortcuts* inside the *model viewport* for those operations that are most common, such as “create a class” or “create a generalization”.

[1]See Sec. 3.1 to view a list of current modeling tools.
[2]In order to strengthen the existing dependency between our CMA and the CASE tool it runs onto, we name “Platform Tool” to this modeling tool.

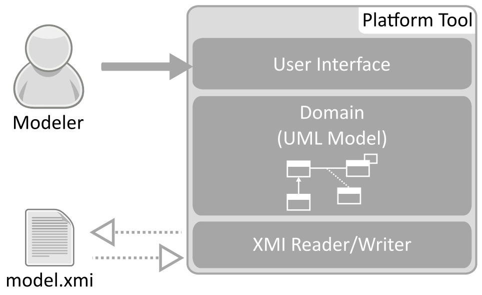
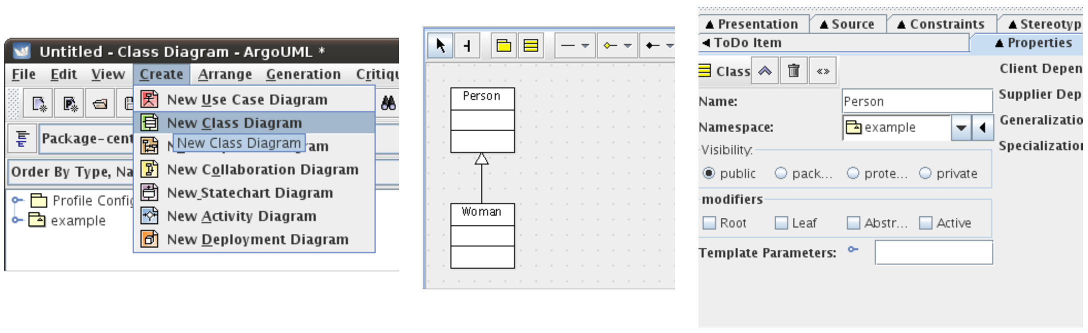


Figure 4.2: Schema of a Platform Tool's architecture.

- *Panels* that show additional information within the application's main window, or *dialogs* that are opened when requested.



Name: Person
Namespace: example
Visibility: [✓ public] pack... prote... private
modifiers: Root Leaf Abstr... Active
Template Parameters:

(a) Menus and tool bars.

(b) Viewport and shortcuts to common actions.

(c) Additional properties panel.

Figure 4.3: Examples of a Platform Tool's GUI.

The data layer stores and retrieves models from a database or a file system. It is important to know which solution is used, in general, by Platform Tools to save information about models, because the CMA defines additional information that has to be also saved somehow, such as the critics that have to be addressed. From our previous experience, we can assume that almost all Platform Tools work with files.

Finally, the logic layer is responsible for capturing, analyzing the feasibility, and processing all the commands triggered by the modeler. Moreover, it usually keeps all the information of the current model loaded from the data layer, such as the existing entity types, or the relationships between them, as well as additional data that are necessary for the correct operation of the application. That is, this layer defines the concrete behaviour a Platform Tool has.

As long as the CMA relies on the Platform Tool, it is very important to know *which* features are usually implemented in these tools and, moreover, *how* they are implemented. In Sec. 3.1, we have seen a few examples of these features, like code generation, reverse engineering, XMI support, documentation generation, among others. However, there are a couple of features we have not seen yet, which are specially important for our CMA. These features are: (1) consistency handling and (2) the meta-commands *undo*/*redo*.

Consistency handling (1) is a basic requirement to get correct models. UML defines several consistency rules, which may or may not be implemented by Platform Tools. Moreover, those Platform Tools that implement them do it in different ways. Therefore understanding the behaviour of a Platform Tool when dealing with a potentially inconsistent environment becomes difficult. For example, there is a consistency rule that states that "there can not be two classes with the same name". While some Platform Tools do not allow the creation of two classes with the same name, and so they avoid the inconsistency, others do.

The truth is, though, that it does not matter whether the Platform Tool implements consistency handling or not. If modelers are interested in this functionality, and the Platform Tool they use does not include it, it can be included inside the CMA; consistency rules could be programmed as a CMA feature and, whenever inconsistencies are introduced, the CMA can outline them. However, as we shall discuss later in Sec. 6.3 the Platform Tool has to implement some UML constraints.

The meta-commands *copy/paste* and *undo/redo* (2) are included in the vast majority of Platform Tools. While the former commands can be seen as an "automatic creation of objects", and can thus be treated like any other regular command, the latter ones require special attention.

As we shall see in Sec. 6.3, it is mandatory to know which solution was implemented within a concrete Platform Tool in order to properly adapt the CMA. Usually, an *undo/redo* mechanism follows a *linear undo*, which uses a chain of commands where only the previous and next commands can be undone or redone [4]. To our knowledge, this mechanism can be implemented using one of the following techniques:

Commands Each time the modeler issues a *command*, this command is stored in a *commands history*. Each *command* knows how to *perform* an action and how to *undo* it. When the modeler moves backwards, the proper command is retrieved and its *undo* method called. When she moves forwards, the *execute* method of the proper command is called instead. See Sec. A.2.3 for a detailed explanation of this solution.

Snapshots Each time the modeler issues a command that changes the model, a *snapshot* of the model is taken. A *list of snapshots* is kept and, whenever the modeler navigates back and forth through the snapshots history, the current model is replaced by the proper snapshot. This solution is very fast, at the expense of increased memory consumption.

4.3 The CMA

The CMA has to meet the following non-functional requirements:

- It has to integrate as many *guidance* and *assistance* features as possible within a Platform Tool,
- these features may be either a *new functionality* that automates a certain task, or a *criticism* that outlines some "error" within the model that has to be improved,
- we want these new features to be easily included into the CMA; in other words, we want the CMA to be *extensible*,
- the CMA *has to run on top of existing CASE tools*, which means that it supports some kind of adaption to the Platform Tool it runs onto, and
- this adaption has to be done in a seamlessly manner; the CMA has to fit its Platform Tool's behaviour as best as possible.

Fig. 4.4 illustrates the previous ideas. The CMA, which is a piece of software that includes new functionalities and criticisms, requires a Platform Tool to run. In order to interact with it, the CMA defines a couple of Application Programming Interfaces (API), which provide an abstract way to handle the underlying Platform Tool. CMA's new features use these APIs instead of the Platform Tool's, so that they operate properly regardless of the latter's specific implementation.

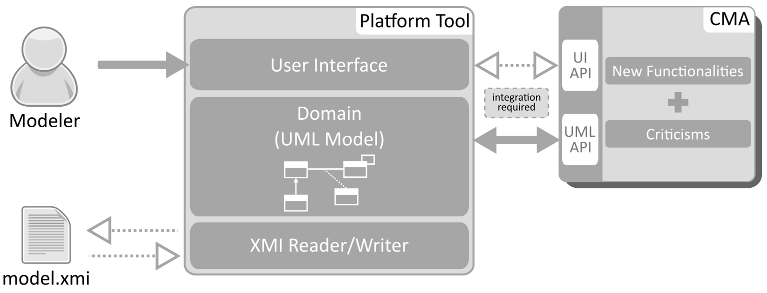


Figure 4.4: The CMA over a Platform Tool.

UI API CMA functionalities and criticisms have to be available and visible to the modeler, so the UI has to be modifiable somehow. The User Interface API provides a set of functions to create menus, dialogs, etc. Another benefit obtained by using an API allows a tiny integration with the Platform Tool: when the CMA is adapted to a specific Platform Tool, and this API is thus implemented, the resulting UI items are seamlessly integrated with the ones found in the Platform Tool.

UML API New functionalities need to access the current UML model. In Ch. 1, we have seen that a UML model is an instance of a UML meta-model. This API provides an implementation of the UML meta-model, so the concrete UML meta-model implementation found in the Platform Tool can be wrapped and, thus, decoupled from the CMA. It provides a set of operations to create, modify, delete, and query UML elements.

5 CMA's Architecture

The goal pursued by the CMA is the improvement of the model's quality. In order to achieve it, the CMA provides *criticisms* and *new functionalities* that should be included in current modeling tools. In this chapter, we present the architecture we propose to cope with this goal.

First, we present an overview of this architecture. This overview includes the definition of the *Command* and *Task* concepts as the materialization of *new functionalities* and *criticisms*. We also discuss how the complexity our CMA has to deal with can be simplified by organizing components in two different levels: the *operational level* and the *knowledge level*.

Second, we describe the APIs by which the CMA can be adapted to a specific Platform Tool. On the one hand, the *UML API* provides a mechanism to access and modify the Platform Tool's UML model. On the other hand, the *UI API* offers a few components to modify the Platform Tool's User Interface, so that the CMA can, for example, interact with the modeler.

Finally, we describe in detail the *operational* and *knowledge levels*. The former tracks the consequences of executing commands in the Platform Tool. These consequences may be the creation of tasks pointing out a defect introduced by the command, or the finalization of tasks that do no longer apply, because the defects they were pointing out disappeared. The latter, on the other hand, has all the required information to handle the operation and evolution of the *operational level*: in other words, it maintains the definitions of what can be criticized, how *Tasks* evolve, and which *new functionalities* (*Commands*) can be introduced.

5.1 Architecture Overview

In this section we present an overview of the CMA's architecture. We first present and describe the concepts of *Commands* and *Tasks* as the implementations of *new functionalities* and *criticisms* used throughout the first chapters of this master's thesis. Then, we show the complexity our CMA has to deal with: how to include new functionalities, how to detect defects, how to notify the modeler, etc. In order to simplify this complexity, we define a two-level architecture. This architecture organizes the components in two different levels: the *operational level* and the *knowledge level*.

5.1.1 Commands and Tasks

The goal pursued by the CMA is the improvement of the model's quality. In order to achieve it, the CMA includes *criticisms* and *new functionalities*. On the one hand, the CMA *criticizes* the work done by the modeler. Thus, whenever something that requires modeler's attention happens, the CMA notifies her. On the other hand, the CMA avoids reinventing the wheel by offering *shortcuts* to common tasks and *automating* as much work as possible. Our CMA's architecture implements these ideas using *Tasks* to represent *criticisms* and *CMA Commands* to represent *new functionalities*.

A *Task* is anything that needs to be addressed by the modeler so she can improve her model. For example, assume we want class names of a model to follow the following guideline: "they have to start with a capital letter". Whenever a new class is created, or its name is modified, the CMA would check whether the new name follows the guideline. If it does not, a new *Task* stating that "the class c does not follow the naming guideline; the first letter of the name n should be changed so it begins with a capital letter" is created.

A *Command* is anything that can be executed by the modeler and modifies the model's state. Platform Tools already have commands, like "create class" or "modify name" We can integrate new functionalities in the Platform Tool by defining *Commands* in the CMA. These *Commands* are named *CMA Commands*. For example, "automatic attributes gathering" to automatically include attributes in a class based on the class name, or "capitalize the first letter of a class name" so the name follows the guideline.

5.1.2 Operational and Knowledge Levels

The architecture of the CMA has to allow the integration of as many features as possible on top of current modeling tools. If we focus on a couple of examples (*naming* and *code completion*) of those presented in Sec. 4.1, we will notice the complexity of our CMA. On the one hand, a feature concerning naming properties may state that (1) "class names have to begin with a capital letter". Thus, if, for example, the model we are working with (2) "has a class named *person*", the CMA (3) "would notice it" and (4) "would notify the modeler with a task". On the other hand, a model completion feature may include a new command (5) that automatically gathers attributes for a class based on its name. If we execute such a command for a class named *Person*, we would (6) get a list of attributes like, for example, *name* or *birthday*. These examples show common problems that could be controlled and solved by the CMA. Despite their apparent simplicity, we can see that the CMA has to deal with a high level of complexity to include them. In particular, the following questions may arise:

1. How to include a set of "rules" or "guidelines" that define what is "correct" and what is not?
2. Which rules are violated by a model and why?
3. When does the CMA check the model's correctness?
4. How does the CMA notify the modeler?
5. How does the CMA include new commands?
6. How does the CMA compute the effects of a command?

These questions have to be answered by our CMA's architecture. If we find a solution that successfully answers them in an efficient manner, we will be able to build a CMA that achieves our goals. In order to organize this complexity, we use the ideas Fowler presented in [18] when describing the *Accountability Pattern* (see Sec. A.1.1), and we thus structured the architecture in two levels: the *knowledge level* and the *operational level*.

The *knowledge level* includes the *general knowledge* that describes which properties a model has to have to be correct, and when this correctness may be violated. For example, it may contain a property stating that "class names have to start with a capital letter" (1). This property may be violated "whenever a class is given a name" or "its name is changed" (3), because the new name may not start with a capital letter. This knowledge would also be able to describe that a new command, such as the *automatic attributes gathering*, is available (5). The information this level maintains is independent from any model; it is something we always know, regardless the models we work with.

The *operational level* includes the *concrete knowledge* of a model. That is, which commands were issued to modify the model, which properties the model violates, and which command produced those violations. For example, we may know that we issued a command to change the name of a certain class. If the new name was "person", we would violate the guideline defined in the *knowledge level.* Thus, this level is strongly related to the model we are working with and the *knowledge level.* On the one hand, it points out the defects that *this* particular model has. On the other hand, it is the *knowledge level* which defines "what a defect is", "when it may be introduced", etc.

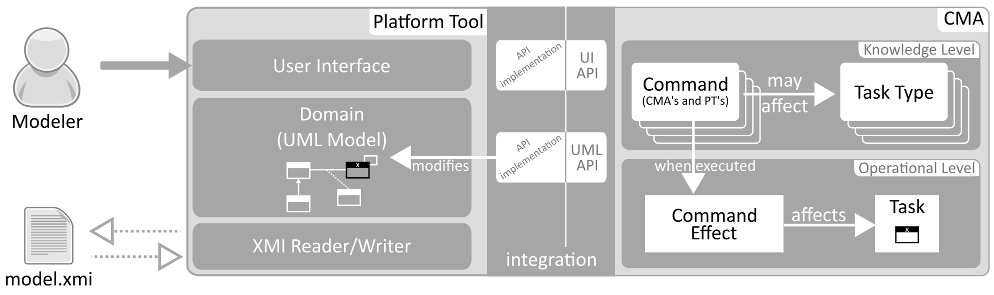


Figure 5.1: Detail of the CMA's architecture.

Figure 5.1 shows a detailed view of the CMA's architecture. We can see that the CMA is adapted to a Platform Tool by implementing its APIs. The *knowledge level* knows which *Commands* are available, and that these commands, when executed, *may* introduce or correct defects in the model. The *operational level* maintains information on what has actually happened. When a *Command* is executed, it affects the defects the model has: it creates new defects, or it corrects (some of) the existing ones.

5.2 The APIs

In this section, we define the UML and the UI APIs. These APIs have to be implemented by a Platform Tool if we want the CMA to run on top of it.

5.2.1 The UML API

As we stated in Ch. 1, UML models are instances of the UML meta-model. Since Platform Tools work with UML models, these tools are supposed to implement the UML meta-model. In Ch. 4, we briefly explained that the CMA needs to access the model maintained by the Platform Tool. The problem is that every Platform Tool has its own implementation of the UML meta-model, making it impossible for the CMA to manage those concrete implementations directly.

In order to overcome this problem, we applied the *Adapter Pattern* described in Sec. A.2.2. When the CMA is adapted to a concrete Platform Tool, an implementation of this API has to be provided, adapting the Platform Tool's UML meta-model. Thus, Platform Tool's instances are wrapped by the implementation of the CMA's API. Figure 5.2 shows the simplified version of the UML meta-model our CMA uses. The UML API, which is detailed in Appx. B, is comprised of a set of operations scattered among the classes of this meta-model.

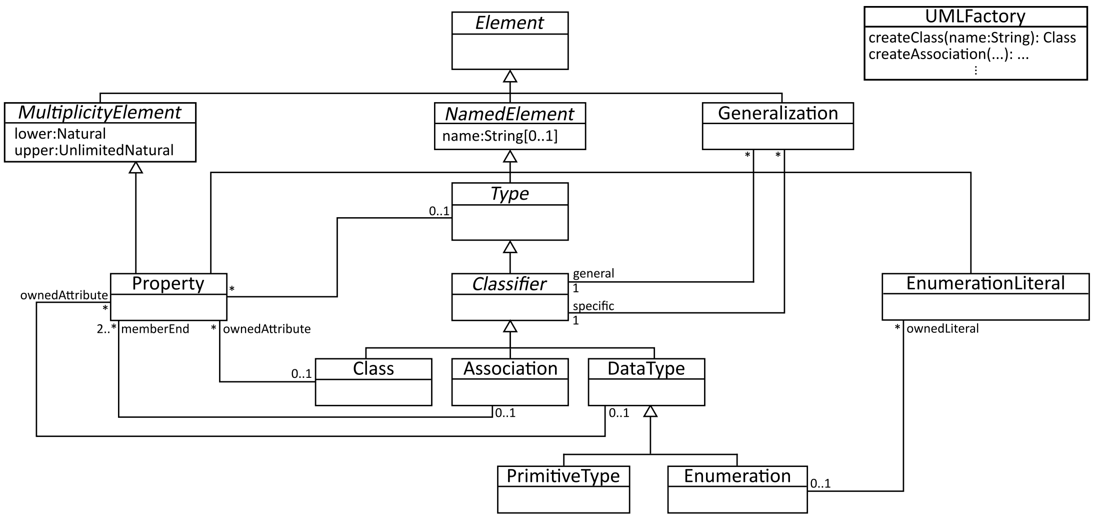


Figure 5.2: The UML API is a simplified UML meta-model.

In Appx. B, we can see the complete documentation of all the operations defined in the API. The definition of this API is based on the MDT-UML2Tools for Eclipse [14]: the operations are distributed among the different UML elements in order to simplify its usage. For example, in order to create an attribute inside a class C, we have to invoke the *createOwnedAttribute* method from C. Additionally, the API provides two more classes to operate with UML diagrams: a *UML Factory* class and a *UML Utilities* class.

The *UML Factory* implements the *abstract factory pattern* described in Appx. A.2.1. The implementation provided by the Platform Tool of our API instantiates UML elements using the meta-model defined in the Platform Tool, but the returned instances are properly wrapped using our own API. Since this factory is part of the API, we can assure that its implementation matches the underlying Platform Tool. As a result, whenever a new UML element is created using this factory, a Platform Tool's UML element is instantiated and it is properly adapted to the implementation of our API. The *UML Utilities* class simplifies the retrieval of information. It provides a set of functions to access those UML elements that are already defined in the model.

Note that the classes defined in this API, which follow the *adapter pattern*, do also follow the *factory pattern*. When invoking an operation from one of those classes to instantiate new UML elements, a Platform Tool's UML elements is instantiated, and a wrapped version of it is returned.

5.2.2 The UI API

The adaptation of the CMA to a Platform Tool does also require some UI integration. The CMA has to be able to define new *Menus* to show the new *Commands* it provides, to interact with the modeler to query her information, and to show the *Pending Tasks* it is maintaining. Figure 5.3 shows an extremely simple UI API that allows us to perform these tasks.

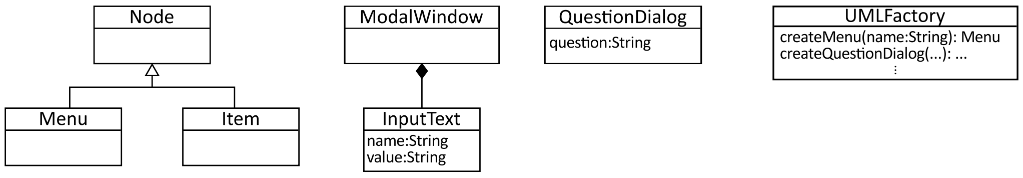


Figure 5.3: The UI API includes some UI elements required to interact with the modeler.

The UI API includes the definition of *Menus*, which can include more *Menus* and *Items*, and some elements to interact with the modeler. The *QuestionDialog* is a UI element whose purpose is to ask a yes-or-no question to the modeler. The *ModalWindow*, on the other hand, implements a form that displays a set of *InputFields* and expects the modeler to fill them in with some values.

5.3 Operational Level

The *operational level* is closely related to the way the Platform Tool operates. Generally, commands change the state of the model; this level tracks the consequences of executing these commands. These consequences may be the creation of tasks pointing out a defect introduced by the command, or the finalization of tasks that do no longer apply, because the defects they were pointing out disappeared.

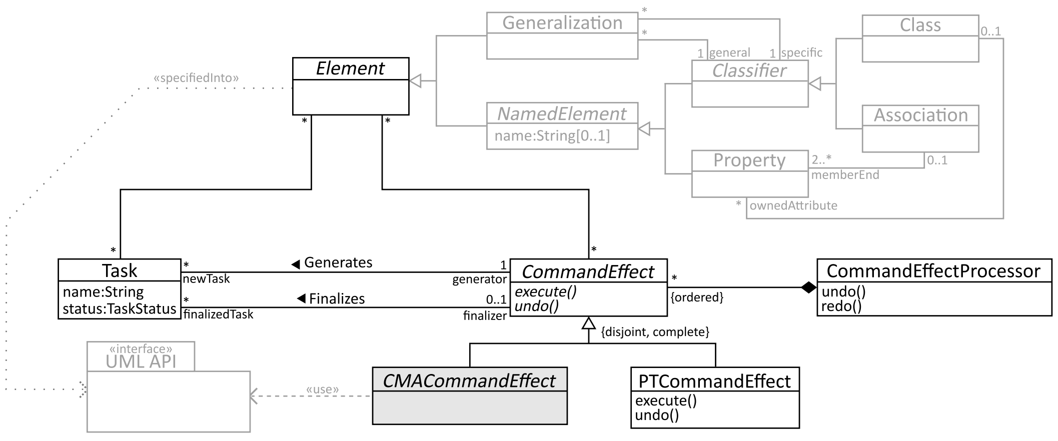


Figure 5.4: Conceptual schema of the CMA's *Operational Level.*

Figure 5.4 illustrates the main components of this level. A *Task*, whose goal is to point out a defect in the model, is generated and finalized because of the effect of a command execution (named *CommandEffect*). For example, if the modeler issues a *SetName* command, and she changes the name of a class to "person", an "invalid capitalization" *Task* would be generated by this *CommandEffect* and would be related to the *Element*[1] (in this particular case, a *Class*) that has the invalid name.

5.3.1 Command Effects

We have seen that a *Command* is whatever the modeler can execute in order to modify model's state. When the modeler issues a command from a UI menu, a *CommandEffect* is created and its *execute* method is called. Since we are interested in improving the quality of a model, we only care about those commands that modify the model. The commands that do not change it, such as "list classes" or "computing the importance of a class", are not of this level's interest.

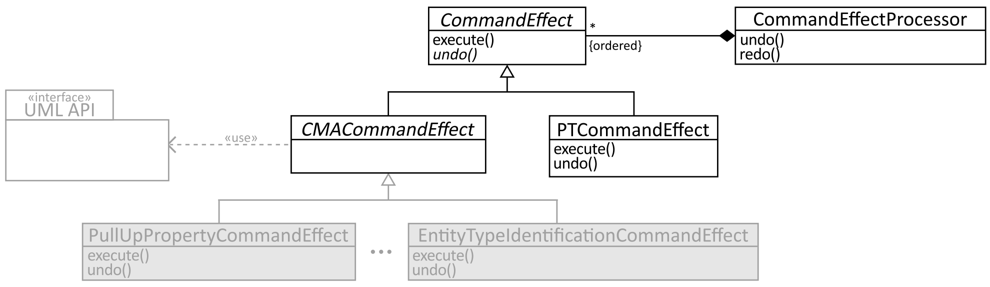


Figure 5.5: Detail of the *Command Effects*.

Figure 5.5 shows two types of *CommandEffects*: *Platform Tool's* and *CMA's*. The former are those commands that the Platform Tool offers to the modeler, regardless of the CMA. Some typical examples of this kind of commands are the creation of a new class, the creation of an association, or the deletion of an attribute. The latter are those commands that were added to the Platform Tool by the CMA, such as "automatic attributes gathering". Both types follow the *Command Pattern*, with an undo/redo mechanism, but their behavior is different:

- If the command issued is defined in the Platform Tool, we need to do almost nothing at CMA level. The CMA only requires to know that a certain command was issued because, as we have said, it needs to check, and eventually criticize, its consequences.
- If the command is defined in the CMA, its effect has to be coded in the CMA. In order to provide new operations to the modeler, we need to extend the *CMA Command Effect* class. A new operation has to provide the code required to perform its effects. Since a *CMA Command Effect* follows the Command Pattern, with an undo/redo mechanism, both the *execute* and the *undo* methods have to be implemented.

5.3.2 Tasks

A *Task* is anything that needs to be addressed by the modeler so her model can be improved: if the model changes, and the changes introduce "defects" to the model, a *Task* pointing them out is created. In other words, the goal of a *Task* is the notification of defects a model has. The introduction of tasks to our system requires some sort of management, which involves the creation and finalization of tasks. This *Task Management* leads us to the following questions:

- When and why is a task created?

[1]See Fig. 5.2.

- When and why is a task finalized?

At the *operational level*, we can answer *when* a task was created or finalized, and *what* created or finalized it. However, as we shall see, in order to provide an answer to *why* a task was created or finalized, we require additional information, which is not available at this level, but at the *knowledge level*.

Tasks' Behavior

As we can see in [5, 15], an approach to detect inconsistencies in real time consists on detecting when the model changes and, then, detecting what has been changed. Following this idea, in order to instantiate new *Tasks*, which can be thought as our "inconsistencies", we just need to detect when a *Command* was issued, because it is the only way to change the model.

It is expected that any *Task* in the system can be solved; that is, the defect it points out can be corrected. It is the modeler's job to fix the tasks pointed out by the CMA. In order to achieve it, the modeler must execute a set of one or more commands, so the model changes and, ultimately, the task becomes done.

Figure 5.6 illustrates the *Task*'s state diagram. When a new task is created, its state is *Pending* which means that the task requires some action from the modeler to be solved. When the modeler continues her work, executing commands while she is modeling, it might be the case that one of those issued commands resolves the task, either by *completing* it or by *canceling* it.

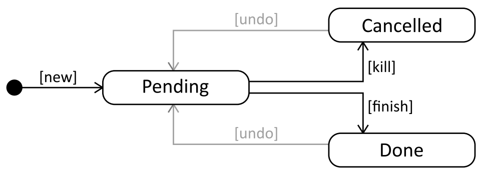


Figure 5.6: *Tasks*' State Diagram.

The *Done* and *Canceled* statuses both represent that the task does no longer apply, because the defect it pointed out no longer exists. The difference is subtle, but important: a *Task* becomes *Done* when the *Command* fixes the problem the task outlined, whilst it becomes *Canceled* if there is a new *Task* pointing out the same defect. For example, renaming a class named "person" to "Person" sets the "capitalization task" to *Done*, because the new name follows the capitalization guidelines. Now, assume we have a class named "person". Since this name does not follow the guideline, the CMA has a *Task* pointing out this defect. If we rename the class to "man", a new *Task* has to be generated, because the new name does not follow the guideline either, and it has to *cancel* the previous one.

In principle, it is not possible to change a *Task* state from *Done* or *Canceled* back to *Pending*. Once a specific task has been *Done* or *Canceled*, it remains with this state forever. The truth is, though, that the state of a "non-pending" *Task* can become *Pending* again because of the special command "undo", which leads the model to its previous state.

The association *generates* between a *Command* and a *Task* represents the fact that every single *Task* our CMA maintains is due to the execution of a concrete *Command*. All the tasks linked to a *Command* by this association are in the *Pending* state. The association *finalizes* represents that the task does not apply anymore because the execution of the associated *Command* "corrected" the defect pointed out by a *Task*. As we have seen, these tasks may be either *Canceled* or *Done*.

5.3.3 Structural Events

We have defined that the changes occur in a model because of *Command* executions or, in other words, because of *Command Effects*. The rationale is that the only way to change the model is by executing a command. As a result, if we are able to detect when a command is issued, we are able to detect when changes occur. However, this solution entails some problems. If we use *Structural Events* instead of *Command Effects*, the problems disappear.

An Example of Why Structural Events are Better than Command Effects

In Ch. 4, we have seen that our CMA is an extensible tool. The features that someone can define for and include in the CMA are independent: they can be plugged in and removed from the CMA easily. Assume that two different people, John and Kate, have different CMAs at different places. That is, their CMAs differ on the functionalities each one offers to the modeler.

On the one hand, John added a new functionality to his: the *Pull-Up Property* refactor. This refactor assumes that there is a set of classes which are an specialization of another class; if there is a common property between those subclasses, it is usually a good practice to remove that property from every single specialization and to place it in the superclass.

Kate, on the other hand, included the capitalization feature: she wants to ensure that all the attributes of her model begin with a small case letter. Whenever a modeler invokes the *Create Attribute* command, the CMA checks the rule Kate included and, if it does not hold, creates a new *Task* pointing it out.

Now, suppose that there is a third person, Sawyer, who wants to include both functionalities to his own CMA. Since the two functionalities are independent, there should be no problem including them in our framework. But when Sawyer starts using his CMA, he realizes that if he performs a Pull Up Property from an attribute named "Age", the CMA does not create a *Task* stating that it should be renamed to "age". This happens because when Kate made that feature, the only command that could create an attribute was the *Create Attribute* command; she had no idea that a refactor command named *Pull-Up Property* existed.

This result is unexpected and disconcerting. To solve this problem, Sawyer could adapt the functionality Kate created, and specify that the task Kate defined must also be instantiated whenever a *Pull-Up Property* command is issued. However, this solution goes against one of our goals: whenever someone "installs" a new functionality, it is supposed to work "out-of-the-box".

It is obvious that both the *Create Attribute* and the *Pull-Up Property* commands may require the creation of the task Kate defined, because they both create an attribute, and it might be the case that this attribute's name does not follow the guideline. Hence, we are not interested in the commands itself, but in the *Structural Events* they generate.

Definition

A *Structural Event* is an elementary change in the population of an entity type or relationship type [8]. We have defined the following structural events:

`create` creates a new instance of a UML Element.

`delete` deletes an instance of a UML Element.

`link` creates a new instance of an association that relates the types of the two instances. In order to determine which association is being instantiated, the role names, which are unique, are used.

`unlink` deletes an instance of an association that relates the types of the two instances. In order to determine which association is being instantiated, the role names, which are unique, are used.

`set` sets a value to an attribute defined in the type of the instance.

`unset` removes the value of an attribute defined in the type of the instance.

Example 1. Assume we want to create a new *Class* x named “Person” with an “Natural”[2] attribute y named “age”. The *Structural Events* generated are:

```
create(x[Class])
set(x, name[="Person"])
create(y[Property])
set(y, name[="name"])
link("property", y, "type", n)
link("class", x, "ownedAttribute", y)}
```

Example 2. Suppose we want to delete a *Generalization* g between two classes x and y. The *Structural Events* generated are:

```
unlink("general", x, "generalization", g)
unlink("specific", y, "generalization", g)
delete(g[Generalization])
```

Integration of Structural Events

We have already discussed the necessity of determining *Tasks*’ behavior in terms of *Structural Events*. Since a *Command Effect*, at the end, is just an ordered set of *Structural Events*, the modifications we have to apply to our model are quite simple. When a *Command Effect* is issued, we need to compute the *Structural Events* it triggered and create or finalize the tasks according to those *Structural Events*.

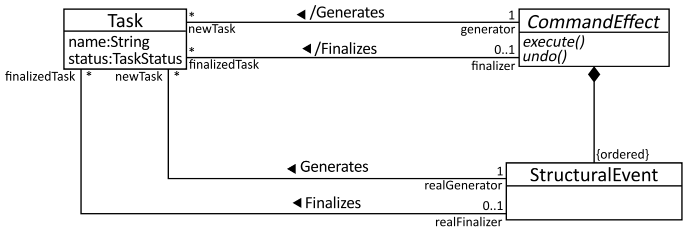


Figure 5.7: Detail of the task generation at the *operational level*.

Figure 5.7 shows the generation of *Tasks* when using *Structural Events*. An *Structural Event* is always related to the *Command* that triggered it. Now, a *Task* is generated or finalized by a *Structural Event*, and the relationship between a *Task* and a *Command Effect* becomes a derived association.

The *Structural Events* we have defined are `create`, `delete`, `link`, `unlink`, `set`, and `unset`. These *Structural Events* can be classified depending on the meta-type they affect. Hence, `create` and `delete` affect meta-classes, such as the instantiation of a *Property*; `link` and `unlink` affect meta-associations, such as relating a *Property* to a *Class*, so the former becomes

[2]Assume that the “Natural” *Type* is the variable n.

an attribute of the latter; and `set` and `unset` affect the meta-attributes of the UML meta-classes, such as defining the *name* of a *Class*, or the *upper* value of a *MultiplicityElement.* In Fig. 5.8 we have defined three subclasses of *Structural Event* which correspond to these three categories.

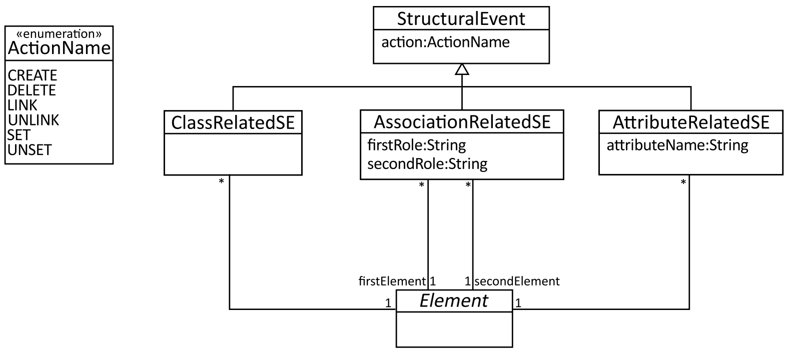


Figure 5.8: Detail of *Structural Events* (*operational level*).

There is an integrity constraint which states which *ActionNames* value can be set to each subclass of *Structural Event*: *Class Related Structural Events* can only use *Actions* `create` and `delete`; *Association Related Structural Events* can only use `link` and `unlink`; and *Attribute Related Structural Events* can only use `set` and `unset`.

How to Determine the Structural Events a Command Effect Produces

In order to determine the *Structural Events* a *Command Effect* produces, we use different approaches based on the concrete type of the *Command Effect. Platform Tool's Command Effects* rely on the proper adaptation of the CMA; that is, the Platform Tool has been properly modified to generate the *Structural Events* associated to every single command it defines. On the other hand, *CMA's Command Effects* rely on the *Proxy Pattern.* The CMA accesses and modifies the Platform Tool's UML model via our UML API. When the CMA is adapted to a concrete Platform Tool, the API is implemented for that particular tool, and thus the CMA can access the information the tool handles. Sticking to our API ensures that the execution of its operations conform to the expected behavior. Two different API implementations have to behave exactly in the same manner. This characteristic allows us to determine in advance which structural events will be issued by the operations of our CMA.

Hence, by using the *Proxy Pattern*, our CMA provides its own implementation of the API which, in turn, uses the specific Platform Tool's implementation. Every time an operation is executed in our CMA's API implementation, the Platform Tool's is invoked and then the *Structural Events* are properly generated.

5.4 Knowledge Level

The *operational level* tracks the consequences of the execution of every single command issued by the modeler. It maintains the tasks that point out the defects introduced by the structural events a command generated. It also maintains those tasks that were corrected or canceled, and the command that finalized them. The *knowledge level* has all the required information to handle the operation and evolution of the *operational level.* Whatever is defined at the *operational level* has an *alter ego* at the *knowledge level* that describes its behavior.

This section begins with a brief description of what a *Command* is and its relation with a *Command Effect*. We then discuss the concept of a *Task Type* and its relation with *Tasks*. We also explain what *Structural Event Types* are, and how they are used to describe the behavior of a *Task*; that is, when a *Task* is created, completed, or canceled. Figure 5.9 shows the conceptual schema corresponding to the CMA's *knowledge level*, which includes all these concepts. For the sake of simplicity, this figure hides some information which is shown in figures 5.10 and 5.13.

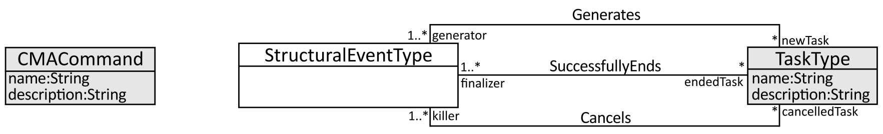


Figure 5.9: Conceptual schema of the CMA's *Knowledge Level*.

5.4.1 Commands

We already know that a *Command* is whatever the modeler can execute in order to modify model's state. We have seen that the execution of a command generates a *Command Effect* in the operational level. This effect executes the code that actually performs the modifications to the model's state. Moreover, the effect is stored in a *Command Effect Processor* to allow the modeler to navigate back and forth the executed command and, thus, provide the "undo" functionality. We also know that there are two types of *Command Effects*, depending on where they are defined: *CMA Command Effects* and *Platform Tool Command Effects*.

When a Platform Tool, along with the CMA, is started, it needs to know all the available operations; otherwise, they could not be shown to the modeler and, thus, she would not be able to execute them. Since the CMA can define new operations, the concept of a *CMA Command* has to be modeled somewhere so, ultimately, the Platform Tool can detect which commands are additionally available and can show them to the modeler. As *CMA Commands* are independent from the current working model, they are defined in the knowledge level[3].

5.4.2 Structural Event Types and Task Types

The goal *Tasks* pursue is outlining the defects a model has. The set of *Pending Tasks* the modeler has to address evolves according to the *Structural Events* generated. But, how does the CMA know that "when a certain *Structural Event* is triggered, a certain *Task* has to be generated[4]"? In order to answer this question, we have to define two concepts and a relationship between them: the *Structural Event Type* concept, the *Task Type* concept, and the *generates* association. Thus, we are able to say that "whenever a *Structural Event* happens, the *Tasks* generated are those whose type is related to the type of the *Structural Event*".

The creation of these *types* entails additional problems that have to be addressed:

- Provided that *Structural Events* occur regardless of *Structural Event Types*, how do we determine the type of a *Structural Event* once it has been triggered?

[3]Note that this differs from what we first presented in Fig. 5.1. At the knowledge level, we only need to define the *CMA Commands*, not Platform Tool's. A Platform Tool already knows the *Commands* it provides. However, the CMA has to "tell the Platform Tool" which are the new commands it provides.

[4]For the purposes of simplicity, from now on and during this section, instead of talking about *generating*, *completing*, and *canceling* a *Task*, we will be only referring to *generating* tasks. However, whatever we say about the *generates* operations does also apply to the other two operations.

- When a certain type of *Structural Event* occurs, it *may* generate a new *Task*. We know this because we have an association relating the type of the *Structural Event* and the type of the potentially creatable *Task*. How do we know that the *Task* has to be actually generated?
- When a *Task* is generated, it points out the offending *Elements*. How does the CMA know which *Elements* have to be related to this new *Task*?

Determining the Type of a Structural Event

Figure 5.8 presented the different *Structural Events* that can be triggered by the Platform Tool or by our *CMA Commands*. The *Structural Events* where classified in three groups: *Class*, *Association*, and *Attribute Related Structural Events*. Each group was related to the *Element* instance it affected (an *Association Related Structural Event* affects two instances).

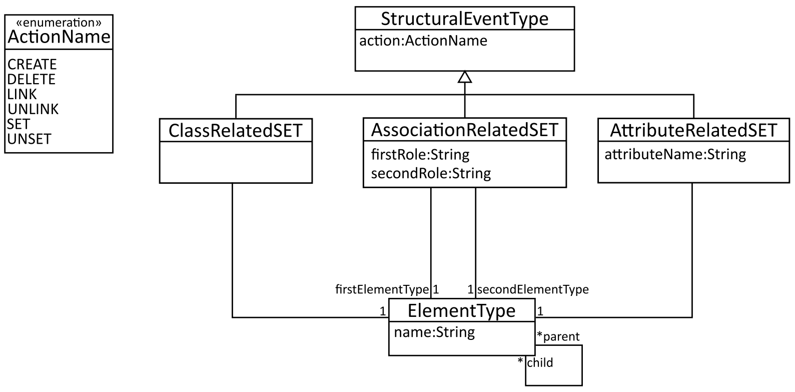


Figure 5.10: Detail of *Structural Event Types* (*knowledge level*).

Figure 5.10 shows the *Structural Event Types* our CMA is able to deal with. As we can see, the three groups presented in the *operational level* are preserved. The only difference is that they are no longer related to an *Element*, but to an *Element Type*. Using this structure we are able to state predicates like "whenever a *Class* is *created*, it may be the case that a *Task* of a certain *Task Type* has to be generated", or "whenever a *Named Element* has *its name set*, it may be the case that a *Task* of another specific *Task Type* has to be generated".

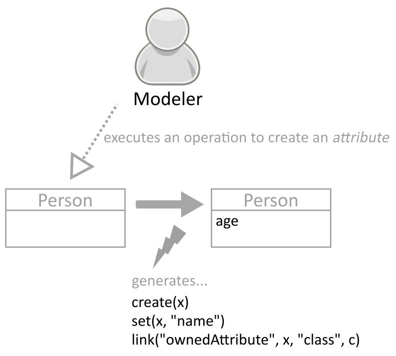
In order to determine the *Structuarl Event Type* of a *Structural Event*, we need to take into account information about the *Structural Event* and the *Elements* it is related to. We define a *candidate type* of a *Structural Event* as the *Structural Event Types* that match the following properties:

- If the *Structural Event* is a *Class Related Structural Event*:
  1. The *actions* of both the *event* and the *type* match.
- If the *Structural Event* is an *Association Related Structural Event*:
  1. The *actions* of both the *event* and the *type* match.
  2. The *firstRoles* of both the *event* and the *type* match.
  3. The *secondRoles* of both the *event* and the *type* match.

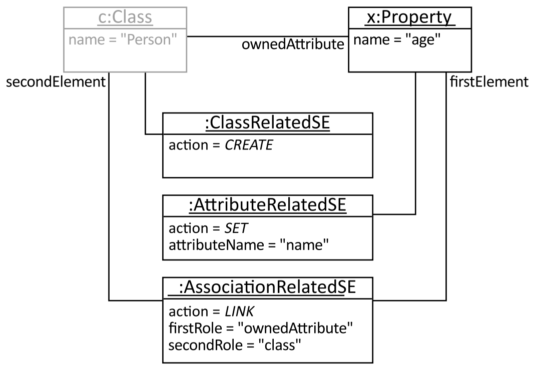
- If the *Structural Event* is an *Attribute Related Structural Event*:
  1. The *actions* of both the *event* and the *type* match.
  2. The *attributeNames* of both the *event* and the *type* match.

Finally, to determine which *candidate types* are actually a type of a *Structural Event*, we look at the *Elements* and *Element Types* to which they are related:

- If the *Structural Event* is a *Class Related Structural Event*, or an *Attribute Related Structural Event*, we only need to check that the type of the *Element* to which the *Structural Event* is the same to which the *candidate Structural Event Type* is related, or a subclass.
- If the *Structural Event* is an *Association Related Structural Event*, we have to perform the same check with the two elements it is related to. The type of the *firstElement* has to match the *firstElementType* to which the *candidate Structural Event Type* is related to, and the type of the *secondElement* to the *secondElementType*.


Modeler
executes an operation to create an *attribute*
Person
Person
age
generates...
create(x)
set(x, "name")
link("ownedAttribute", x, "class", c)

(a) Structural Events generated by an attribute creation.



(b) Object Model of these Structural Events.

Figure 5.11: Example of *Structural Events* generation.

Consider the example shown in Fig. 5.11. Suppose the modeler decides to add an attribute x named *age* to the class *Person*. When she executes the command to perform this creation, a set of *Structural Events* is generated. Their *Structural Event Types* are, to mention a few:

- `create(x)`:
  - *create(Property)*
  - *create(NamedElement)*
  - ...
- `set(x, "name")`:
  - *set(Property, "name")*
  - *set(NamedElement, "name")*
  - ...

- `link("ownedAttribute", x, "class", c)`:
  - *link("ownedAttribute", Property, "class", Class)*
  - *link("ownedAttribute", Property, "class", Classifier)*
  - ...

By using this solution, we can now define defects that apply to more than one *Element* type. For example, we can define a rule stating that "whenever a *NamedElement* is given a name, check if the name is properly spelled; if it is not, generate a *Task* stating that "the name seems to be misspelled"", which applies to all *NamedElements* regardless of its concrete type, like a *Class*, a *Property*, or an *Association*.

Event-Condition-Action Rules

Suppose we have a *Structural Event Type* x related to a *Task Type* t. If an x-type *Structural Event* is triggered, it *may be* the case that a t-type *Task* has to be generated. In order to generate the *Task* once the *Structural Event* has been received, a certain *Condition* has to hold.

The rules that handle the way our *Tasks* evolve consist of three parts: the *event* part, which specifies a list of events, the *condition*, which is a query on the UML model, and the *action*, which in our case is the generation of one or more *Tasks*. These rules are known as Event-Condition-Action (ECA) rules[10].

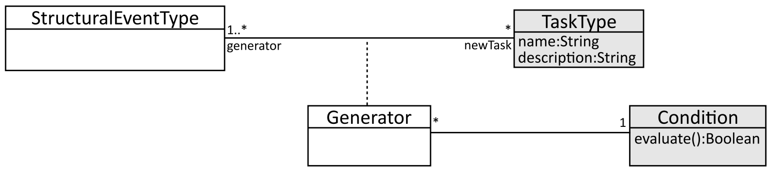


Figure 5.12: *Conditions* included in the relationship between *Structural Event Types* and *Task Types*.

In Fig. 5.12 we can see how we include the *Conditions* in our model. To make it simpler, only the *Generator* association is shown. As we can see, the relationship between a *Structural Event Type* and a *Task Type* is now an *Associative Class*, in order to include further information. This associative class is related to the *Condition* governing the actual generation of *Tasks* of a certain type when a *Structural Event* occurs. When the modeler includes a new *Task Type* in the CMA, it needs to provide the code of the *Condition* that evaluates whether a *Task* has to be actually created or not.

How to Implement Conditions

Tasks are related to the offending elements. That is, if we have a *Class* named "person", we would have a *Task* pointing out that *this* class does not follow the naming guideline we set. As we have seen, the *knowledge level* has *Structural Event Types*, *Conditions*, and *Actions*. How did the CMA link that *Task* to the offending class "person"? Or, more precisely, if a *Condition* only states whether a defect was introduced, how does the CMA know which are the offending elements?

Suppose there are some offending elements in the model, and that a *Task* has to be generated. In other words, we have received a *Structural Event*, and the associated *Condition* evaluates to *true*. Since the *Structural Event* is related to an *Element*, we may think that this element is the offending one. In the previous example, when we set the name of a class to "person", the CMA would have received an *Attribute Related Structural Event* related to the class, the evaluation of the *Condition* would have returned "true", and, as a result, a new *Task* related to the offending class could have been created.

The problem is that this solution does not always work. For example, suppose we want to detect duplicate class names. If we created a class named "Person", and there already was one class with that name, the CMA would have to create a *Task* related to the two offending classes. However, the *Structural Event* received is only related to the class we have created, not to the already existing one.

Figure 5.13 illustrates the solution to overcome this problem. A *Condition* can be either a *Generator Condition* or a *Finalizer Condition*. The former evaluates the model and, if a *Task* pointing out an error has to be created, it then returns the set of offending *Elements*. The latter, on the other hand, evaluates whether a *Task*, to which it is related, has to be finalized or not.

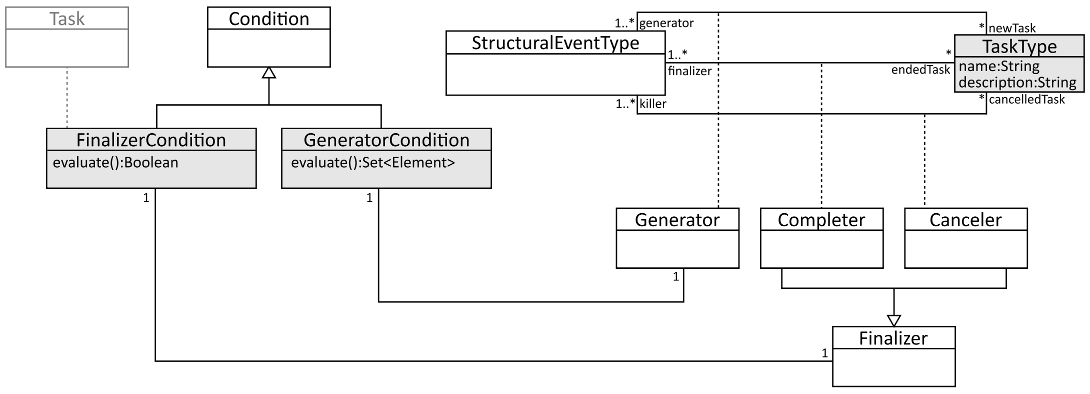


Figure 5.13: Detail of *Conditions*.

How to Finalize Tasks

We have been describing how *Tasks* are generated and set to *Pending*, but not how they are set to *Done* or *Canceled*. The only difference between the *Generator* association and the other two is how the *Conditions* are implemented. Whilst the *Generator* association is related to a *Generator Condition*, which returns the set of offending elements when a *Task* has to be generated, the other two are related to a *Finalizer Condition*.

A *Finalizer Condition* is invoked in order to determine whether tasks of a certain type have to be finalized. When a *Structural Event* is received, the CMA retrieves those *Tasks* whose type corresponds to the *Task Type* to which the *Structural Event Type* is related and, for each *Task*, it evaluates whether the *Condition* holds or not. If it does, the *Task* becomes *Canceled* or *Done*, depending on the concrete associative class the condition was related to. When a *Structural Event* is received, this process is repeated for each *Task* of the associated type.

5.4.3 Further Assistance

Tasks evolve because of *Structural Events*. We have seen that if a *Task* is generated, the only way to finalize it is by generating another *Structural Event*. However, there are some *Tasks* that can not be finalized by any *Structural Event*; these *Tasks* require the intervention of the modeler. For example, suppose we have the following naming guideline for *Class* names:

The name of a *Class*

- should be a noun phrase, whose head is a countable noun in singular form,
- written in the Pascal case (that is, every word in the phrase begins with a capital letter), and
- if N is the name of the *Class*, then the following sentence has to be grammatically well-formed and semantically meaningful:

  An instance of N is [a|an] `lower(`N`)`.

  where `lower(`N`)` is a function that gives N in lower case and using blanks as delimiters.

Whenever we set the name of a class, a *Task* querying the modeler if the sentence makes sense is created. This *Task* cannot be successfully finished by a *Structural Event* (it could be canceled if the class was removed).

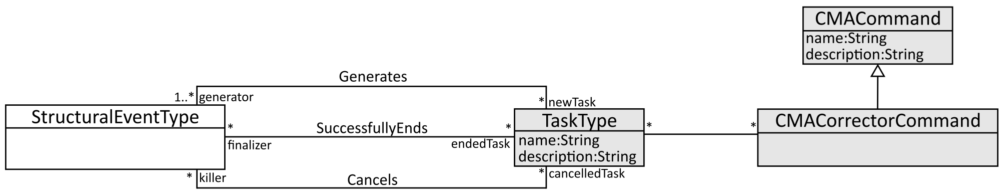


Figure 5.14: *CMA Corrector Commands* included in the *knowledge level.*

In order to solve this problem we define a special type of *CMA Command* whose purpose is to finalize this kind of tasks: *CMA Corrector Commands.* As we can see in Fig. 5.14, *Task Types* can be finalized by executing one of the *CMA Corrector Commands* they are linked to.

Once we have introduced this concept, it is easy to see that we can provide assistants that guide the modeler in the process of finalizing *Tasks.* For example, suppose we have a *Task* pointing out that there is a class whose name is "person". We can provide a *CMA Corrector Command* that, when executed, automatically changes "person" to "Person".

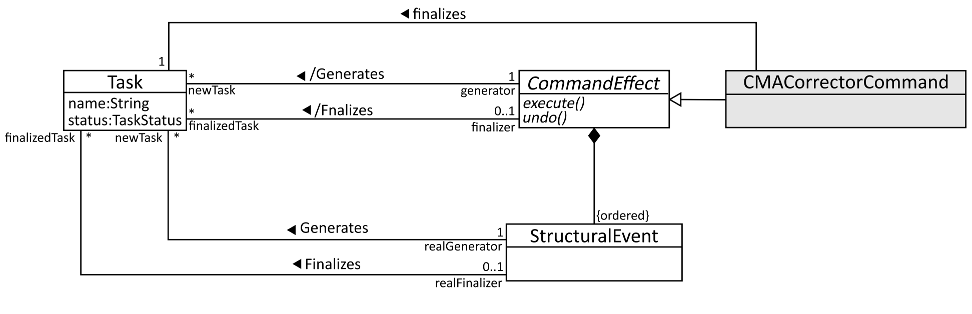


Figure 5.15: *CMA Corrector Command Effects* included in the *operational level.*

Finally, note that we have to modify the *operational level* to include these new type of *Commands.* Figure 5.15 shows the adopted solution. We define *CMA Corrector Command Effects* as a subclass of the already defined *CMA Command Effects* class, and a relationship *Finalizes* between the *corrector* and the *task.*

6 Construction of a Prototype

In this chapter we describe the construction of a prototype. The prototype requires (1) a Platform Tool, (2) the CMA, and (3) adapting the CMA to the Platform Tool.

First, we present the design and implementation of a Custom Platform Tool. Since the adaptation of the CMA to a specific Platform Tool requires detailed knowledge on how the tool is implemented, we decided to build an extremely simple modeling tool to be used as our prototype's Platform Tool.

Second, we detail the implementation of the CMA's architecture. This implementation is organized using the following two modules: the *core module* and the *plugins module*. The former comprises the minimum knowledge (which includes the *knowledge* and the *operational levels*) required for the correct CMA operation, whilst the latter populates the *knowledge level* in order to add new features and detect new defects.

Finally, we describe the adaptation of the CMA to the Custom Platform Tool. This adaptation involves (1) the implementation of the UML and UI APIs defined in the architecture, (2) the modification of Platform Tool *Commands* to generate *Structural Events*, and the modification of the Platform Tool's UI to (3) show *CMA Commands* and (4) which *Pending Tasks* and *Task Types* are available.

6.1 Design and Implementation of a Custom Platform Tool

In order to create a running prototype we need a Platform Tool to which our CMA can be adapted. The adaptation to a specific Platform Tool requires detailed knowledge on how the tool is implemented. It is not the aim of this thesis to study and understand the technical details of a *real* Platform Tool, which would be a preliminary step if we want to adapt the CMA on it. However, in order to implement a running prototype, having a Platform Tool is a requirement. In this section, we explain the design and implementation of an extremely simple modeling tool to be used as our prototype's Platform Tool.

6.1.1 Platform Tool's Architecture

If we take a look at Fig. 4.2, we can see Platform Tools usually have a three-layered architecture: a *User Interface* layer, a *Domain* layer, and a *Data* layer to load/save models from/to files. In order to define our own Platform Tool, we followed such a layered architecture. Figure 6.1 provides a detailed view of it, showing the main components located at each layer.

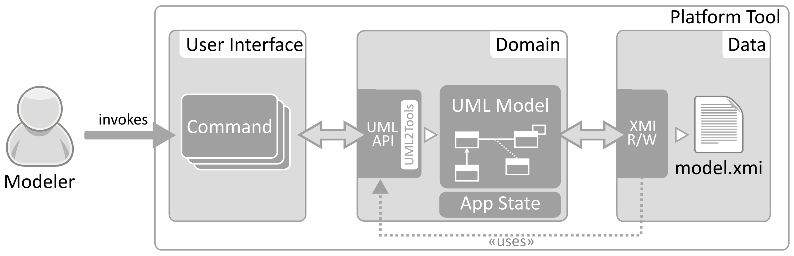


Figure 6.1: Architecture of our Platform Tool.

The User Interface (UI) Layer is how the user interacts with the system. It presents to the modeler a set of *Commands* she can execute in order to modify the model and to query information about it. Thus, a *Command* is “whatever the modeler can execute”. In this case, *Commands* are operations aimed to *modify* and to *query* the state of the UML model.

The Domain Layer maintains the information of the *UML Model* the modeler is working with, and additional *State* information required by the tool for its correct operation.

As we shall see, *UML Models* are defined and handled by using the UML2Tools package. This package is an Eclipse’s plugin, but it can be used in a stand-alone fashion.

The *State* information our Platform Tool handles comprises the stack of undos and redos available and the name of the XMI file we are working with, among others.

The Data Layer saves and retrieves UML models to and from files. Basically, it is a module that provides a couple of operations to work with XMI files.

Figure 6.2 shows the conceptual schema of our Platform Tool. We can see that *Commands* are presented to the modeler using a set of *Menus*, and that these *Commands* use the UML2Tools package to query and modify the UML model. The *PTSystem* class maintains the *State* information. It also offers some operations to list *UML Elements*, because the UML2Tools package lacks such operations. The *UML2ModelSerializer* includes the operations required to work with XMI files.

6.1.2 Implementation

The implementation of our Custom Platform Tool is quite simple. We only need to code the conceptual model shown in Fig. 6.2. As we have seen in Sec. 3.1, modeling tools usually use Graphical User Interfaces. Implementing GUIs like the ones found in a real Platform Tool takes a lot of time, and it is not a goal of this master’s thesis either. Hence, our Platform Tool uses a minimal GUI, which only displays three lists: *Classes*, *Associations*, and *Enumerations*. Whenever an item of one of these lists is selected, its description is shown (Fig. 6.3).

The interaction with the modeler is done using a Command Line Interface (CLI). A CLI is a mechanism for interacting with an application, by which the user types commands to perform specific tasks. For example:

```
> new class
New Class Form
  Name: > Person
  Is it abstract? (y/N) > N
> ls classes
```

```
List of classes
  class Person
  end
```

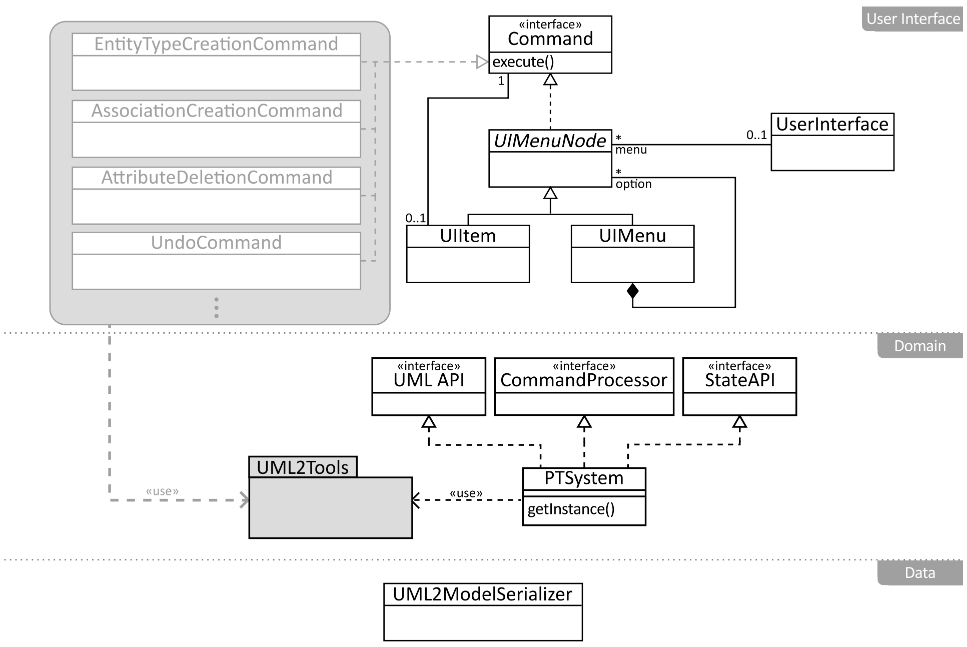


Figure 6.2: Conceptual schema of our Platform Tool.

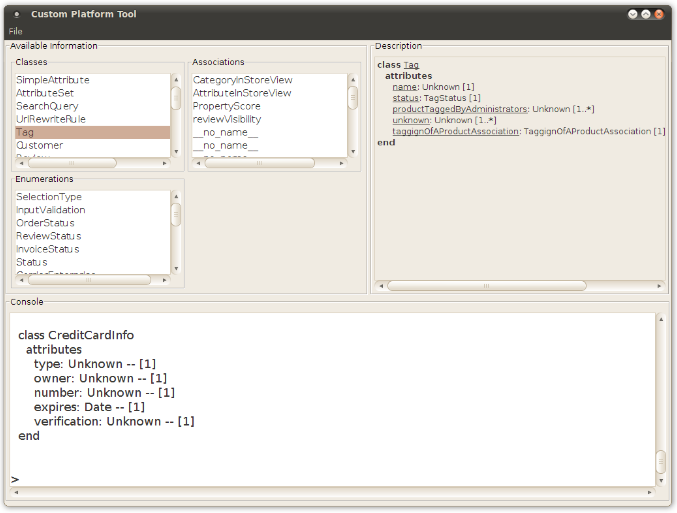
The undo/redo mechanism

Our Platform Tool implements a snapshot-based undo mechanism. Whenever a *Command* is executed, a snapshot of the previous state is generated. The Platform Tool maintains a list of the available snapshots, so the modeler can move back and forth.

6.2 Implementation of the CMA

When we presented the architecture of the CMA we saw that it is organized in two levels: the *knowledge level* and the *operational level.* The implementation we propose, however, is organized using the following two modules: the *core module* and the *plugins module.* The former comprises the minimum knowledge required for the correct CMA operation, whilst the latter populates the *knowledge level* in order to add new features and detect new defects. Hence, the *core* includes the *CMA* class; information about *Structural Events* and *Structural Event Types*; *Commands* and *Command Effects* to maintain their execution; and the interfaces of *Tasks*, *CMA Command Effects*, etc., by which the CMA is extended. The *plugin* component, on the other hand, is responsible for loading new features: it loads the relationships between *Structural Event Types* and *Task Types*, with the associated *Conditions*; and *CMA Commands.*

In this section, we describe the implementation of these two components. First, we focus our attention on the most important part of our system's *core*: the *CMA system class.* It is in charge



```
SimpleAttribute
AttributeSet
SearchQuery
UrlRewriteRule
Tag
Customer
```

```
CategoryInStoreView
AttributeInStoreView
PropertyScore
reviewVisibility
__no_name__
__no_name__
```

```
SelectionType
InputValidation
OrderStatus
ReviewStatus
InvoiceStatus
Status
```

```
class Tag
  attributes
    name: Unknown [1]
    status: TagStatus [1]
    productTaggedByAdministrators: Unknown [1..*]
    unknown: Unknown [1..*]
    taggignOfAProductAssociation: TaggignOfAProductAssociation [1]
end
```

```
class CreditCardInfo
  attributes
    type: Unknown -- [1]
    owner: Unknown -- [1]
    number: Unknown -- [1]
    expires: Date -- [1]
    verification: Unknown -- [1]
end


>
```

Figure 6.3: Screenshot of our Custom Platform Tool.

of maintaining the *operational* and *knowledge* levels, and the API specific implementations provided by the adaptation of the CMA over the Platform Tool.

Second, we explain the implementation of the *knowledge* and *operational levels*. This implementation is straightforward, because we only need to code the UML diagrams presented throughout the previous chapter. We need to take special care with the following classes: the *Command Effect Processor*, which implements the undo/redo mechanism; the *CMA Command*, which is used to add new operations to the Platform Tool; and the definition of the *ECA Rules*, that is, how we define the relationships between *Structural Event Types*, *Task Types*, and *Conditions*.

Next, we detail one of the most important parts of the prototype: the link between the previous two levels. That is to say, we show how we implemented the *Structural Event Processor* class, which is responsible for modifying the *Pending Tasks* set because of the *Structural Events* generated according to the rules defined in the *knowledge level*. Finally, we explain how we implemented the *plugin* component.

6.2.1 The CMA System Class, or how to Prepare the CMA to be Adapted to a Platform Tool

The first thing we need to do in order to build our CMA on top of a Platform Tool is to define a *System Class* which holds all the information our CMA manages. That is, we need a class to maintain the following information:

- which *CMA Commands* are available,
- which *Task Types* the CMA handles and, thus, the defects it is capable to detect,
- which *Structural Event Types* can generate a *Task* of a certain type and under which *Conditions*,
- which specific API implementations have been provided by the Platform Tool for the CMA integration,

- which *Tasks* are associated to the current model, that is to say, which *defects* our model actually has, and

- which *Command Effects* produced these *Tasks*.

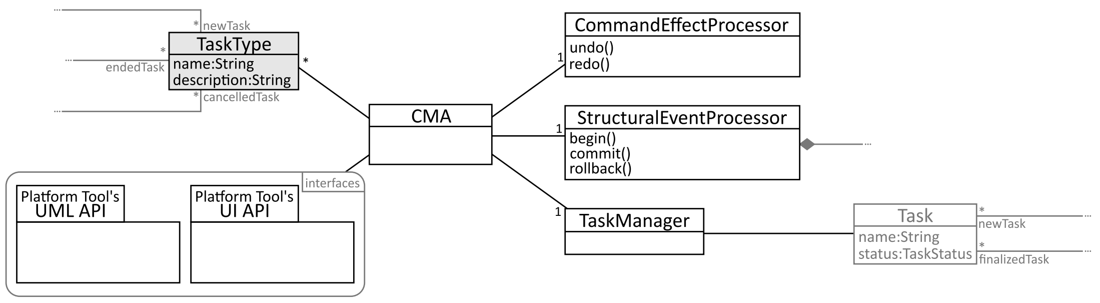


Figure 6.4: UML Diagram of the *CMA System Class*.

The previous information comprises what we presented in the *operational* and *knowledge* levels. Figure 6.4 shows how we tie everything. We can see that there is a *CMA* class which is related to everything else: the *Command Effect Processor*, a *Task Manager*, a *Structural Event Processor*, and the available *Task Types*. It also illustrates that the *CMA* is related to the Platform Tool's API implementation.

CMA's own API Implementation

In Sec. 5.3.3, we said that *CMA Command Effects* invoked an implementation of the API provided by the *CMA*. This specific implementation follows the *Proxy Pattern*: when a *CMA Command Effect* invokes an operation of the API, the implementation it is using is the CMA's own API implementation. When the *CMA* class is instantiated, it requires the instances of the Platform Tool's API implementation: the *UML Factory*, the *UML Utilities*, and the *UI Factory*. The UML instances are wrapped by the CMA's API implementation.

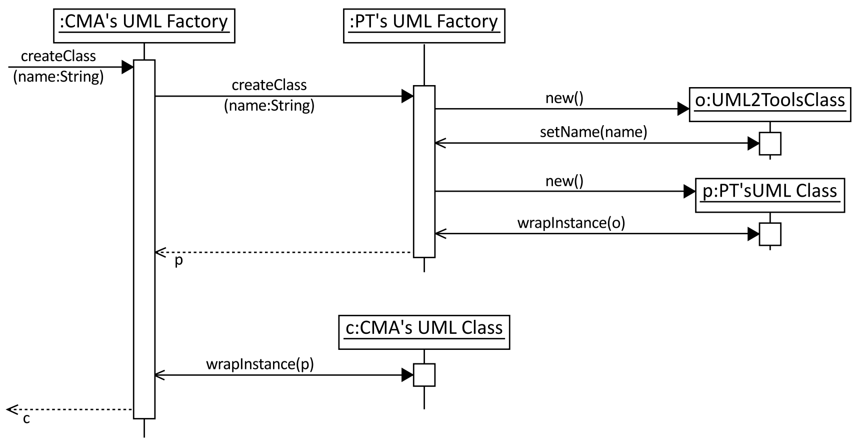


Figure 6.5: Example of how the different API implementations are related and "wrapped".

Figure 6.5 provides an example of how the APIs are related and wrapped. When we create a new *Class* using the CMA's *UML Factory*, the Platform Tool's *UML Factory* is invoked to create the *Class*. The returned *Class* is an instance of the Platform Tool's API implementation. However, since classes can create additional elements, such as *Properties* that will become their attributes, the Platform Tool's *Class* also has to be wrapped to a CMA's *Class* before the CMA's *UML Factory* can return its result. Now, if someone invokes the `createOwnedAttribute` defined in the returned *Class*, it will invoke the `createOwnedAttribute` defined in the Platform Tool's *Class* and, then, the associated *Structural Events*.

6.2.2 Operational and Knowledge Levels

The implementation of these levels is straightforward. However, we believe that there are some classes that require special care: *Command Effect Processor*, the *CMA Command*, and the definition of the *ECA Rules*. These components are important because they define how the CMA matches the underlying Platform Tool and how it can be extended.

The Undo/Redo Mechanism

In Ch. 4 we presented two different solutions to implement a *linear undo mechanism*: *Commands* and *Snapshots*. The former solution assumes that the changes in a model occur because of the execution of a *Command*. Every time a *Command* is issued, a new instance of its execution is saved. Since a *Command* execution knows the changes it produces, it also knows how to undo them. The latter solution, on the other hand, assumes that each time the model is changed, a snapshot of its previous state is generated. Thus, whenever we want to recover a previous state, we just replace the current model with the appropriate snapshot. As we stated, our CMA has to be able to work with both solutions and adapt itself to the specific solution implemented in the Platform Tool.

When the modeler undoes an operation, she wants to undo all the effects it caused. That is, she wants the model to be as it was in the previous state and, if the CMA is running on top of the Platform Tool, she also wants the *Tasks* to be as they were in the previous state. At first, it may seem that, in order to restore *Tasks* to their previous state, it is not necessary to know which undo mechanism is implemented in the Platform Tool. Our *operational level* "knows" that an action *generated* and *canceled* a specific set of *Tasks*. By using this information, we can undo the changes the action produced to the *Tasks*. However, we do need to know the specific approach used by the Platform Tool, because the CMA also includes *CMA Commands* that have to undo the effects they generated in the model.

A command-based undo mechanism matches our implementation of *Commands* and *Command Effects*, because it follows exactly the same idea. Whenever a *Command* is issued, its execution is saved; that is, a *Command Effect* is generated and saved. If the Platform Tool uses this approach, and the modeler undoes an action, the CMA must call the `undo` operation to recover the model's previous state. Otherwise, the *Tasks* would be in their previous state, but the model would not.

A snapshot-based undo mechanism, on the other hand, does not require the `undo` operation of a *Command Effect*. Whenever the modeler wants to undo an action, the previous state is restored from a snapshot, which was taken by and is stored in the Platform Tool. In this case, using *Command Effects* in the CMA does not matter. When an action is undone, the CMA can restore the *Tasks* to its previous state, without calling the `undo` operation.

In short, the difference between these two solutions is whether the CMA calls, or not, the `execute` and `undo` methods of a *CMA Command Effect* when *redoing* or *undoing* actions. *Platform Tool Command Effects* are already handled by the Platform Tool; we only need them to be represented in the *operational level* so we can modify *Tasks* status as explained. However, *CMA Command Effects* have to execute its code if we are in a *command-based undo* mechanism, because, otherwise, the Platform Tool would not be able to restore the previous state.

Note that in a command-based approach, each time an action is undone or redone, the code that modifies the model is executed again. This means that the *Structural Events* are triggered again, too. We have decided that, whenever an action is being undone or redone, the *Structural Events* it generates are ignored, and we use the information between a *Command Effect* and a *Task* to recover the *Tasks*' appropriate state.

CMA Commands

CMA Commands purpose is to notify the Platform Tool that new operations are available. The code that actually modifies the UML model is defined within a *CMA Command Effect*. Since every single *CMA Command* is related to one *CMA Command Effect*, the only thing our CMA needs to know is this relation. When new commands are introduced in the CMA, a new instance of a *CMA Command* is created, using the provided name, and related to the *CMA Command Effect* class. When a *CMA Command* is *executed*, it creates a new instance of the *CMA Command Effect* and the latter's code is executed.

ECA Rules

In order to implement the model illustrated in Fig. 5.13, we normalized the associative classes (see Fig. 6.6). *Structural Event Types* are related to one *Generator Condition* and two *Finalizer Conditions*. These classes are subclasses of *Condition*, which is related to the *Task Type* it affects. Whenever a new *Task* is supposed to be created, an instance of the class *Task* to which the *Task Type* is "related" is created. Whenever new defects are supposed to be detected by the CMA, new subclasses of *Generator Condition* and *Task* have to be provided. *Finalizer Conditions* or *CMA Corrector Commands* are also required to cancel or end a *Task*.

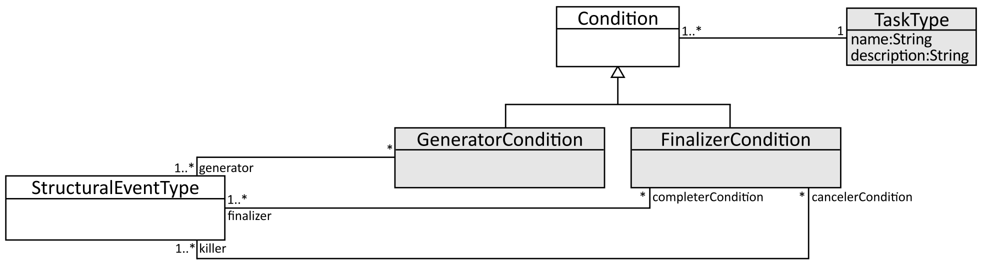


Figure 6.6: Detail of the *Conditions*' implementation.

6.2.3 Tasks Behavior based on Triggered Structural Events

One of the most important parts of our CMA is *Structural Event Management*. We have already seen that *Structural Events* are the backbone of *Tasks* generation and finalization. In this section we describe how *Structural Events* are generated and related to the command that created them. We also describe how we process them to modify *Task* states.

Generating and Packing Structural Events

The execution of a *Command Effect*, whichever its concrete type is (a *CMA Command Effect* or a *Platform Tool Command Effect*), generates a set of *Structural Events. CMA Command Effects* generate these events because of the CMA's API implementation. When the CMA is adapted to a Platform Tool, latter's commands are properly modified to generate the events. Since every execution of a Platform Tool command is represented by an instance of a *Platform Tool Command Effect*, we can link this instance to the generated structural events.

In order to simplify the creation of the *Structural Events* and its link with the *Command Effect* that generated them, we use two classes: *Structural Event Factory* and *Structural Event Processor*. The *Factory* provides a set of operations with the parameters that are required to generate a *Structural Event*. When one of these operations is invoked, the *Structural Event* is generated and sent to the *Structural Event Processor*. The *Processor* analyzes the received *Structural Events* and modifies the *Tasks* according to the information stored in the *knowledge level*.

In order to know which *Command Effect* generated a set of *Structural Events* we need these *events* to be "packed". The *Structural Event Processor* defines three operations to handle the received *Structural Events*: `begin`, `commit`, and `rollback`. When a *Command* is executed, it notifies the *Command Events Processor* by invoking its `begin` method. When the `begin` method of the *Structural Event Processor* is called, the processor is ready to receive *Structural Events*, until the `commit` operation is invoked. When it is, the *Structural Events Processor* analyzes the *Structural Events* and modifies the *Tasks* of the system accordingly. However, if the `rollback` operation is invoked, the received *Structural Events* are discarded, and the *Tasks* processing is aborted.

Processing Structural Events

If an operation invokes the *Structural Event Processor*'s `commit` operation, it means that its execution has finished, and it is therefore time to determine which *Tasks* it affected, if any. That is, we need to check if there are any *Completed*, *Canceled*, and/or *Generated Tasks*. For each *Structural Event* related to that *Command Effect*, the CMA checks if

1. any of the *Pending Tasks* can be completed and, if so, it completes them;
2. any of the *Pending Tasks* can be canceled and, if so, it cancels them; and
3. new *Tasks* have to be generated and, if so, it generates them.

In other words, the CMA instantiates the associations between *Tasks* and *Structural Events* shown in 5.7. The associations between *Tasks* and *Command Effects* are derived; they can be *materialized* or *calculated*. For efficiency reasons, we decided to *materialize* the associations, by "copying" the links between *Tasks* and *Structural Events*. Once the information is materialized in the associations between a *Command Effect* and the affected *Tasks* set, the *Structural Events* and their links can be deleted, because they are redundant. Again, we decide to remove them for efficiency reasons. Thus, *Structural Events* and their associations are "temporary information" used to compute the other associations.

Note that it may be the case that a *Command Effect* generates and finalizes a *Task* within its execution. For example, if we create a *Class* named "Person", the operation triggers two *Structural Events*. The first one, "creation of a class", may generate a *Task* stating that "*Named Elements* need a name". The second one, "set the name of the class", completes the *Task*. When we materialized the associations between *Tasks* and *Command Effects*, we did not materialize the links between a created-and-finalized *Task* and its associated *Command Effect*.

6.2.4 Plugin-based System

We stated our CMA has to be *extensible*. Usually, programs are extensible by using *plugins*. A *plugin* is a piece of software that adds specific capabilities to a larger application, by using a certain API. In our case, we can increase the capabilities of our CMA by adding new *Commands* and new *Tasks*. The addition of a new *CMA Command* involves extending the *CMA Command Effect* class with a new subclass that implements the `undo` and the `execute` operations. On the other hand, new defects can be detected by extending the *Task* and *Condition* classes, and providing the description of which *Structural Event Types* may require the evaluation of a *Condition* to modify *Tasks*.

Following this idea, we propose a *Plugin System*, whose purpose is to populate the *knowledge level*, where each plugin is composed of:

- A file `commands.xml` containing the description of new commands. That is, the name of the *Command* and the class that implements the *CMA Command Effect* interface.
- A file `tasks.xml` containing the description of new tasks. That is, for every new *Task Type* we can detect, its *name*, a brief *description* of what it does, and the set of *Structural Event Types* that affect it. For every *Structural Event*, we provide the name of the *Condition* class.
- The implementations of the new classes.

The next chapter presents a few examples of plugins that could be added to our CMA. For each of them, we present the rationale behind it, the XML files describing the plugin, and the classes that implement our interfaces.

6.3 Adaptation of the CMA over the Custom Platform Tool

The adaptation of the CMA to the Custom Platform Tool requires four main steps:

- Provide an implementation of the UML and UI APIs.
- Modify the *Commands* that were defined in the Platform Tool (see Fig. 6.2) to include the generation of the associated *Structural Events*.
- Provide a wrapper for the *CMA Commands*, so they can be shown in the UI and, thus, be executable by the modeler.
- A mechanism to show the *Pending Tasks* and which *Task Types* are available.

6.3.1 APIs Implementation

In principle, the implementation of the APIs should entail no difficulty. We just need to provide the functionalities defined by the signature of every single operation. In the previous chapter, we said that the UML API implementation provided by the Platform Tool follows the *Adapter Pattern*. Usually, this pattern is implemented by *wrapping* the original instance; that is, a *Class* defined in the Platform Tool (using, as we have seen, UML2Tools), is wrapped to a *PlatformToolClass*, which implements the *Class* interface found in our API. However, this solution has several problems.

Take, for example, the scenario presented in Fig. 6.7(a), where we create a *Class* named "person". Then, we create an attribute named "age". If we undo this action, we need to restore the previous state; that is, we want a model with only one class: "person". Our CMA implements the snapshot-based undo mechanism, which means that, whenever we want to undo the effects of an operation, the whole schema is replaced by a copy of the previous state. This schema replacement means that we change the instances of the schema; they have the same information that was available in the previous state, but the instances are "new". If the creation of the "person" *Class* generated one or more *Tasks*, the wrapped instance of *Class* does no longer match any instance of the model, because they are all new.

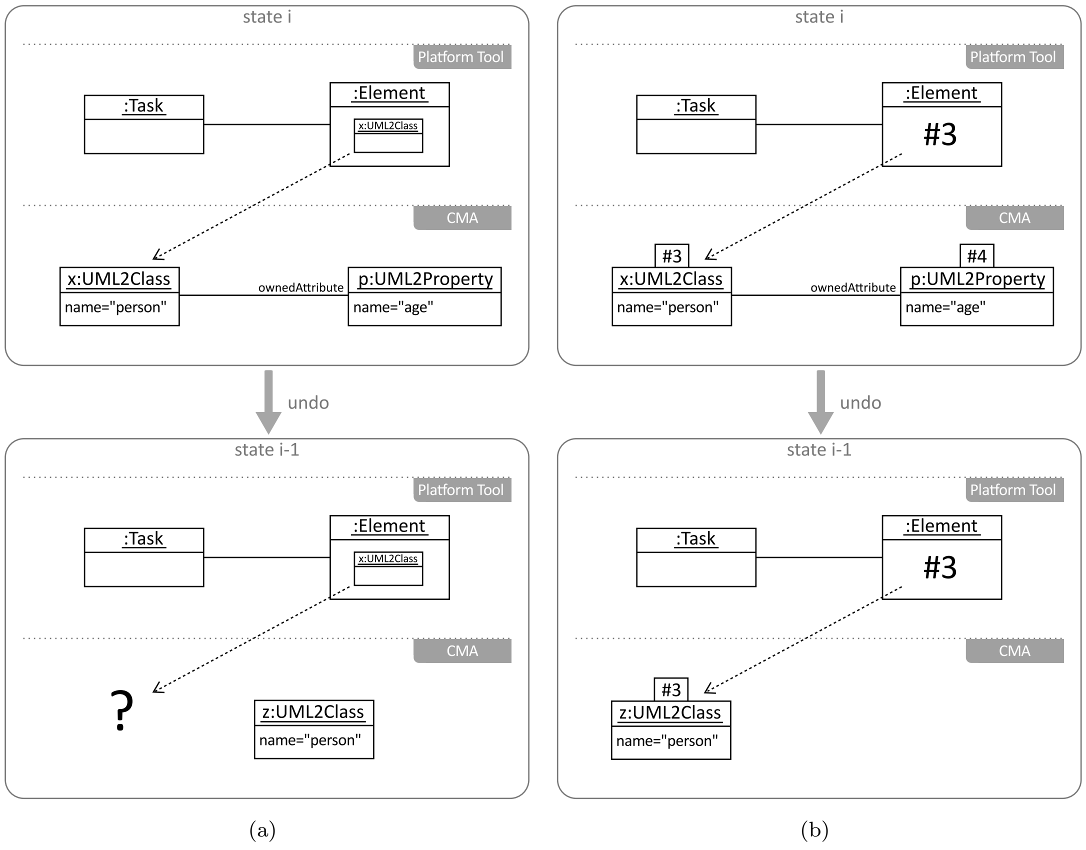


Figure 6.7: Platform Tool's UML API implementation solves potential undo problems.

In order to solve this problem, our UML API implementation does not wrap any UML2Tools *Element*. Instead, we identify these elements with a unique identifier, which is saved in a special *Comment* associated to the *Element* (Fig. 6.7(b). The API implementation maintains the value of this unique identifier. Whenever we want to call an operation of the wrapped *Element*, we look in the model for an *Element* whose identifier is equal to the one our wrapper maintains, and we then call the appropriate operation.

6.3.2 Command Modifications to Include Structural Events

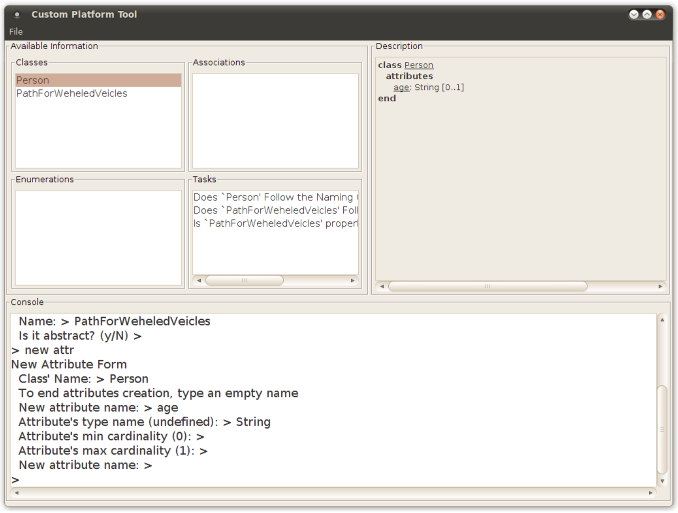
As we have seen in Sec.6.1, the Platform Tool defines a set of *Commands* that can be executed by the modeler. Some of these *Commands* can modify the model, such as the *Association Creation Command* or the *Attribute Deletion Command*, whilst other only query information about it, like the *Enumeration Listing Command*. We need to modify those *Commands* that change the model, so they include the *Structural Events* they generate.

6.3.3 CMA Command Wrappers

CMA Commands must be accessible and executable by the modeler. Otherwise, they would be useless. Since our Custom Platform Tool defines its own *Commands*, which are shown to the modeler via menus, we need to modify the Platform Tool so it can include the *CMA Commands*. In order to achieve this, we decided to use, again, the *Adapter Pattern*. We defined a *CMA Command Wrapper* that wraps an instance of a *CMA Command* to something the Platform Tool can execute.

6.3.4 Additional Tuning

The CMA provides new information which has to be accessible by the modeler: the *Task Types* our CMA controls and the *Pending Tasks* the model has. In order to view this information, we include two new *Commands* in the Platform Tool that access this information and print it in the screen. Moreover, since we decided to include a simple GUI to simplify the interaction with the modeler, we decided to add a new "list" where *Tasks* are shown in real time.



Person
PathForWeheledVeicles

Does `Person' Follow the Naming
Does `PathForWeheledVeicles' Foll
Is `PathForWeheledVeicles' proper

```
class Person
  attributes
    age: String [0..1]
end
```

```
  Name: > PathForWeheledVeicles
  Is it abstract? (y/N) >
> new attr
New Attribute Form
  Class' Name: > Person
  To end attributes creation, type an empty name
  New attribute name: > age
  Attribute's type name (undefined): > String
  Attribute's min cardinality (0): >
  Attribute's max cardinality (1): >
  New attribute name: >
>
```

Figure 6.8: Screenshot of our Custom Platform Tool with the CMA modifications.

7 Experimentation

In the previous chapter we constructed a prototype of our work, which consists on the conjunction of a Platform Tool and the CMA. Since the CMA is extensible, we need to provide plugins to test its correct operation. In this chapter we present some plugins we implemented. We have organized these plugins according to their scope and, for each of them, we describe it, provide some notes on how it was implemented, and show a few results of its execution.

First, we present a *Naming* plugin. The names given to a conceptual schema have a strong influence on the understandability of that schema. This plugin checks whether the names given to the elements of a conceptual schema follow a certain proposal of naming guidelines. Following such a guideline is a first step towards giving good names.

Second, we describe a plugin that checks, to some extent, *Schema Satisfiability.* In Sec. 3.2.2 we presented the concept of *Schema Satisfiability.* A schema S is satisfiable if it admits at least one legal instance of an information base [40].

Next, we propose a naive plugin which provides *Schema Auto-Completion* capabilities. In Sec. 3.3, we presented the Eclipse Platform as an example of an IDE. One of the features IDEs usually include is *code completion.* The idea of this plugin is to follow a similar approach, helping modelers to complete their models automatically. Despite the importance general ontologies have while developing conceptual schemas [11, 47], this plugin does not use them to gather new information: it uses hard-coded information to provide further assistance.

Finally, we present a few conclusions on the work done, valuing the complexity of creating new plugins and adding them to the CMA. We also discuss and compare the different approaches used to implement the plugins presented in this chapter.

7.1 Naming

The names given to a conceptual schema have a strong influence on the understandability of that schema. Giving good names increase its pragmatic quality. However, choosing good names is a very difficult task [44, p.46], and current modeling tools do not help modelers at it.

7.1.1 Description

This plugin checks whether the names given to the elements of a conceptual schema follow a certain proposal of naming guidelines. Following such a guideline is a first step towards giving good names. More specifically, the plugin checks the following properties:

1. Whenever a *Named Element* is created, it should suggest the modeler that she should give a name to the *Named Element*.
2. Whenever a *Class* is given a name, this name
   - should be a noun phrase, whose head is a countable noun in singular form,
   - written in the Pascal case (that is, every word in the phrase begins with a capital letter), and
   - if N is the name of the *Class*, then the following sentence has to be grammatically well-formed and semantically meaningful:

     An instance of N is [a|an] `lower(`N`)`.

     where `lower(`N`)` is a function that gives N in lower case and using blanks as delimiters.
3. Whenever a *Property* is given a name, this name should be a noun phrase written in the Camel case (that is, every word in the phrase begins with a capital letter, except the first one, which begins in lower case).
4. Let E::A:T be an attribute named A of type T of the entity type E. Whenever we give a name to the attribute,
   - if T is not the *Boolean* data type, then the following sentence has to be grammatically well-formed and semantically meaningful:

     [A|An] `lower(`E`)` has [a|an] `lower(`A`)`.
   - but if T is the *Boolean* data type, the name A
     - should be a verb phrase in third-person singular number, and
     - the following sentence has to be grammatically well-formed and semantically meaningful:

       [A|An] `lower(`E`)` may `inf(`A`)`.

       where `inf(`A`)` is a function that gives the infinitive form of the verb in A, and the phrase is written in lower case and using blanks as delimiters.

Some examples of these guidelines are:

1. If we create an *Association* with no name, the CMA suggests the modeler to give it a name (though she can dismiss it).
2. If we create a *Class* named "PathForWheeledVehicles":
   - It has to begin with a capital letter, and
   - the sentence like

     An instance of a *PathForWheeledVehicles* is a path for wheeled vehicles.

     has to be meaningful and grammatically well-formed.
3. If we create an attribute *Name* in a *Class* named *Person*, the CMA warns us that "it should begin with a lower case letter".

4. If we create the attributes *name*:*String* and *studiesArchitecture*:*Boolean* in a *Class* named "Person", the following sentences have to be meaningful and grammatically well-formed:

   A *Person* has a name.
   A *Person* may study architecture.

Furthermore, our plugin also includes a *Spell Checker*. A *Spell Checker* is a program that flags words in a document that may not be spelled correctly. Detecting misspelled words is another step towards our goal: giving good names. Whenever a misspelled word is detected, the CMA should notify the modeler and provide a correct spelling.

7.1.2 Implementation

Whenever the CMA detects there is an *Element* whose name does not follow the proposed guideline, a *Task* has to be generated. As a result, this plugin mainly describes *Tasks*. We have organized *Tasks* in the following categories: *Name Required*, *Capitalization*, *Guidelines*, and *Spell Checking*.

Name Required

Named Elements have an attribute named "name". For some *Named Elements*, like *Classes*, the name is mandatory, whilst for others, like *Associations*, it is optional. We believe that naming all elements in a model is important so, whenever a *Named Element* is created, and if it is not given a name, a *Task* reminding the modeler she should set one is created.

If we take a look at the `tasks.xml` file, we will see that such a *Task* is generated whenever a *Named Element* is created, and it is achieved whenever it is given a name. The *Task* has to become canceled when the *Named Element* is deleted.

There are some cases where a *Named Element* was given a name, but a default one. For example, some Platform Tools set the name of a new *Class* to something like "Class_1". In these cases, it is important to generate a *Task* reminding the modeler that she did not set that name, and so she may be interested in setting one herself.

This *Task* is generated whenever we receive the `set` *Structural Event* for the *name* of a *Named Element*, and the *Structural Event* says the value is *default*. The *Task* is achieved if the *Named Element* is given a non-default value, and canceled if the element is deleted.

Capitalization

As we have seen, *Classes* have to begin with a capital letter. In order to detect whether a *Class* follows this guideline, we define a new *Task* that can be generated whenever a *Class* is given a name. The *Condition* that checks if the guideline is followed needs to validate that the name begins with a capital letter; if it does not, a *Task* has to be generated. However, if the given name follows the guideline, a *Pending Task* would become achieved.

Enumerations and *Associations* follow the same guideline: they have to begin with a capital letter. In order to evaluate whether these two *Named Elements* also follow the guideline, we can use the same *Conditions*. We have to add some new *Structural Events* in the `tasks.xml` file to specify that we also have to check the *Conditions* if the modified name belongs to an *Enumeration* or an *Association*.

Note that the *Condition* ignores if it will check the name of a *Class*, an *Association* or an *Enumeration*. It will be one of them for sure, because the `tasks.xml` file specifies that the *Condition* will be only executed to check the guideline if it is one of them. Since the three types are *Named Elements*, the *Condition* can safely cast the associated *Element* of the *Structural Event* to a *Named Element*.

Following the same idea, we can check the capitalization for *Properties*. The only differences are the concrete implementation of the *Condition*, which now checks whether the first letter of the name begins with low case, and the *Structural Event Types* that may generate or finalize this *Task*, which are related to *Properties*.

Guidelines

The proposed naming guidelines specified (1) how a name had to be written (Camel or Pascal case) and (2) required a sentence using that name to be grammatically well-formed and meaningful. The first condition is already controlled by the previous capitalization-related *Task*. Let's see how to achieve the second one.

We only defined these guidelines for *Classes* and *Properties* that are attributes of a *Class*. Since we are interested in detecting whether their names follow the guideline, the *Structural Event* to track is the `set` *name* of a *Class* and of a *Property*. If we use the same approach we used in the Capitalization category, we would define different *Conditions* and *Structural Event Types* for, on the one hand, *Classes* and, on the other hand, *Properties*. However, we have decided to use the same *Structural Event* and *Condition* to check the guideline. We control the *set Structural Event* of a *Named Element*'s *name*, regardless it is a *Class* or a *Property*. Thus, the *Condition* has to check whether the *Named Element* is a "*Class*" or a "*Property* that is an owned attribute of a *Class*". If it is, it can generate a new *Task* pointing out that the modeler has to check whether the name follows the guideline.

Such a *Task* is canceled if the *Named Element* is deleted. There is no *Structural Event* that can achieve this *Task*. The only way to achieve it is by defining a *Corrector Command*. In the `tasks.xml` file, we specify that this *Task* can be corrected by using a certain *CMA Command Corrector*, which is defined in the `commands.xml`. This *Command* has to generate the specific sentence and query the modeler if the generated sentence makes sense. If it does, the *Task* is achieved. If it does not, it has to remain *Pending*.

Spell Checking

The goal of spell checking is to verify that the introduced names are properly spelled. In order to implement this feature, we have decided to use a new approach. Instead of coding ourselves a spell checker, our plugin queries an external service if the words are properly spelled. This approach becomes very interesting, since we could use external functionalities inside our CMA with little effort.

Whenever a *Named Element* is given a name, we have to check whether this name is properly spelled. Google's Spell Checker is an online service that can correct misspelled words: it receives a phrase, detects which words are misspelled, and provides a suggestion for each of them. Since the spell checker expects a phrase, and we have a single name, like "PathForWheeledVehicles", our plugin has to split the name in one or more words: "Path For Wheeled Vehicles", query the external service, and wait for the response.

Corrector Commands

The *Tasks* presented by this plugin can include a *Corrector Command* that automatizes, and thus simplifies, correcting them. For example, an incorrectly-capitalized noun could automatically change its first letter case to match the guideline; a misspelled name could get a proposal where all the words are properly spelled; etc.

Following this idea, we have define some *Corrector Commands* in the `commands.xml`, and we have modified the `tasks.xml` file to relate these *Correctors* to the specific *Tasks*. In particular, we created three *Corrector Commands*:

- Capitalize the first letter of the name automatically.
- Change the first letter of the name to lower case automatically.
- Provide a correct spell suggestion (via Google).

7.1.3 Results

First scenario

1. The modeler creates three classes: “person”, “man”, and “dona”.
2. The CMA has generated the following tasks:
   - Invalid capitalization of “person”.
   - Invalid capitalization of “man”.
   - Invalid capitalization of “dona”.

```
> new class
New Class Form
  Name: > person
  Is it abstract? (y/N) >

> new class
New Class Form
  Name: > man
  Is it abstract? (y/N) >

> new class
New Class Form
  Name: > dona
  Is it abstract? (y/N) >

> ls tasks
tasks
  - [0] Invalid capitalization of ‘person’.
  - [1] Invalid capitalization of ‘man’.
  - [2] Invalid capitalization of ‘dona’.
```

3. The modeler decides to correct “person”.
4. The CMA offers her to fix the name itself.
5. The modeler accepts the suggestion.
6. The CMA renames “person” to “Person”, and thus a task is corrected.

```
tasks> 0
  Pending task: Invalid capitalization of ‘person’.
  Rationale:
      Normally classes begin with a capital letter. The name
      ‘person’ is unconventional because it does not begin
      with a capital.
      Following good naming conventions help to improve the
      understandability and maintainability of the design.

  Available commands to fix this task:
    [0] Dismiss this task
    [1] Capitalize the First Letter of the Name Automatically

  If you want to correct this task, type the corrector's
  identifier you want to use: > 1


> ls classes
List of classes
  class Person
  end

  class man
  end

  class dona
  end

> ls tasks
tasks
  - [0] Invalid capitalization of 'man'.
  - [1] Invalid capitalization of 'dona'.
tasks> back
```

7. The modeler realizes “dona” is not in English, so she renames it to “woman”.
8. The CMA shows a new task:
    - Invalid capitalization of “woman”.

```
> update class name
Modify the Name of a Class
  Old Name: > dona
  New Name: > woman

> ls classes
List of classes
  class Person
  end

  class man
  end

  class woman
  end

> ls tasks
tasks
  - [0] Invalid capitalization of 'man'.
  - [1] Invalid capitalization of 'woman'.
tasks> back
```

9. The modeler removes the class “man”.
10. The CMA removes the following task:

- Invalid capitalization of “man”.

11. The modeler realizes she made a mistake, and undoes the last action.
12. The CMA restores both the class “man” and the removed task.

```
> rm class
Delete Class Form
  Name: > man

> ls classes
List of classes
  class Person
  end

  class woman
  end

> ls tasks
tasks
  - [0] Invalid capitalization of ‘woman’.

> file undo
> ls classes
List of classes
  class Person
  end

  class man
  end

  class woman
  end

> ls tasks
tasks
  - [0] Invalid capitalization of ‘man’.
  - [1] Invalid capitalization of ‘dona’.
tasks> back
```

Scenario 2

1. The modeler wants to create a new class “Parser”, but when she types the name she misspelled the word and wrote “Praser”.
2. The CMA generates the following task:
   - Is “Praser” properly spelled?
3. The modeler realizes it is not, and queries the CMA if it can provide a solution.
4. The CMA provides “Presser” as a solution, which is not what the modeler wanted.
5. The modeler types herself the name: “Parser”.

```
> new class
New Class Form
  Name: > Praser
  Is it abstract? (y/N) >
```

```
> ls tasks
tasks
  - [0] Is 'Praser' properly spelled?

tasks> 0
  Pending task: Is 'Praser' properly spelled?
  Rationale:
      Google's spell checking software, used by this task, checks
      whether your name uses the most common spelling of  a given word.
      If it thinks you're likely to be wrong, it provides a different
      spelling.

  Available commands to fix this task:
    [0] Dismiss this task
    [1] Provide a Correct Spell Suggestion (via Google)

  If you want to correct this task, type the corrector's
  identifier you want to use: > 1

  Modify Name
  (Type '!cancel' at any time to quit this dialog.
    New name (Presser): > Parser
```

6. The modeler wants to create a subclass of "Parser" named "XMLParser", but she misspells the word again, and writes "XMLPraser".
7. The CMA generates the following task:
   - Is "XMLPraser" properly spelled?
8. The modeler realizes it is not, and queries the CMA if it can provide a solution.
9. This time, the CMA provides a valid solution: "XMLParser".
10. The modeler accepts the suggested solution.

```
> new class
New Class Form
  Name: > XMLPraser
  Is it abstract? (y/N) >

> ls tasks
tasks
  - [0] Is 'XMLPraser' properly spelled?

tasks> 0
  Pending task: Is 'XMLPraser' properly spelled?
  Rationale:
      Google's spell checking software, used by this task, checks
      whether your name uses the most common spelling of  a given word.
      If it thinks you're likely to be wrong, it provides a different
      spelling.

  Available commands to fix this task:
    [0] Dismiss this task
    [1] Provide a Correct Spell Suggestion (via Google)

  If you want to correct this task, type the corrector's identifier
  you want to use: > 1
  Modify Name
  (Type '!cancel' at any time to quit this dialog.
    New name (XMLParser): >

> ls classes
List of classes
  class Parser
  end

  class XMLParser
  end
```

7.2 Schema Satisfiability

In Sec. 3.2.2 we presented the concept of *Schema Satisfiability*. A schema S is satisfiable if it admits at least one legal instance of an information base [40]. For some constraints, it may happen that only empty or non-finite information bases satisfy them. In conceptual modeling, the information bases of interest are finite and may be populated. Then, a schema is *strongly* satisfiable if it admits at least one nonempty and finite legal instance of the information base. Otherwise, we consider that schema incorrect.

7.2.1 Description

In [40], Olivé describes a method to examine whether a schema with a set of cardinality constraints is strongly satisfiable. This method is able to deal with two cardinality constraints that we can define for binary relationship types.

Figure 3.6(b) shows an example that is not strongly satisfiable. In order to verify that no nonempty finite population of the four types (two entity types and two relationship types) satisfies the cardinality constraints, the method builds a directed graph G and checks it does not contain cycles of a particular type. Figure 7.1 shows the graph corresponding to this example. There are two archs for each participant in a relationship type: one from the relationship type to the participant entity type, and the other in the opposite direction.

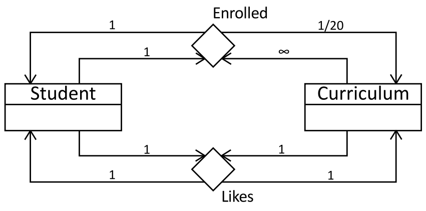


Figure 7.1: The graph G corresponding to Fig. 3.6(b).

Each arch has a weight, which is computed as follows. Let $R(p_1 : E_1, p_2 : E_2)$ be a binary relationship type with cardinalities $Card(p1; p2) = (min_{12},\ max_{12})$ and $Card(p2; p1) = (min_{21},\ max_{21})$. The arch from R to E_1 has a weight w_{12}, where

- $w_{12} = \infty$ if $min_{12} = 0$;
- $w_{12} = 0$ if $min_{12} = \infty$;
- $w_{12} = 1/min_{12}$ otherwise;

The arch from E_1 to R has a weight max_{12}.

It is obvious that G contains cycles. A *critical cycle* of G is a nonempty sequence of archs (v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k) such that

- $v_0 = v_k$, and
- $v_1, \ldots, v_k$ are mutually distinct, and
- the product of the weights of the arcs (v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k) is less than 1.

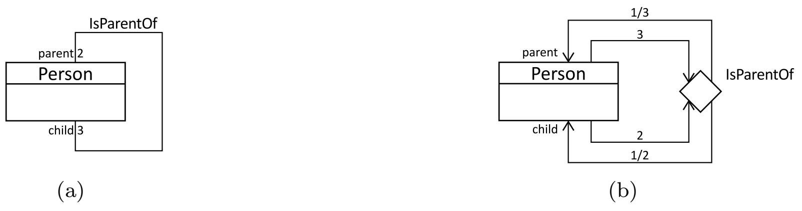


Figure 7.2: A recursive relationship type with non-satisfiable cardinality constraints.

This method can also be applied to recursive types. An example, taken from [40, p.90], is shown in Fig. 7.2. The schema 7.2(a) includes the constraints that each *Person* has to have two *parents* and three *children*. The corresponding graph 7.2(b) has a critical cycle, which proves that the schema is not strongly satisfiable.

7.2.2 Implementation

This plugin only implements one *Task*: "Unsatisfiable Schema because of *some reason*". The "reasons" it is capable to detect are the ones we presented here: binary associations between two classes and recursive types.

Binary Associations

As we have seen, if there is more than one association relating two concepts, the schema can become unsatisfiable because of the cardinality constraints associated to the member ends of each association. The only way a satisfiable schema can become unsatisfiable, according to what the method is capable to detect, is if

- a member end of an *Association* relating two classes A and B has its *lower* or *upper* value modified, and
- there are more associations relating A and B.

In short, we have to (1) detect the "*set*" *Structural Event* of the *upper* or *lower* value of a *Property*, (2) determine if this *Property* is a member end of an *Association* x, and (3) check whether the two classes related by this *Association* have more *Associations* linking them. When these three conditions hold, we have to run the method for every pair of *Associations* that include the *Association* x in order to determine whether the schema is satisfiable or not.

The first step is described in the `tasks.xml` file, where we describe which *Structural Event Types* may generate certain *Task Types.* Steps (2) and (3) are coded inside the *Generator Condition*; if they hold, the same *Generator Condition* calls a `isStronglySatisfiable` method defined in `StrongSatisfiabilityChecker.java`.

Once we have an unsatisfiable schema, the only way it can become satisfiable again is by either

- removing one of the conflicting *Associations*, or
- change one of the multiplicities and hope the new cardinality constraints make the schema satisfiable.

Recursive Entity Types

A similar approach is followed to detect if the schema is satisfiable when we modify the multiplicities of a *Recursive Entity Type*. In order to detect whether the schema is unsatisfiable because of a recursive type, we only need to add an "exception" to the algorithm defined before. Once we (1) detected the *Structural Event*, and (2) determined the *Property* is a member end of an *Association*, we only need to modify step (3) so it checks whether the two classes related by the *Association* are the same; if this is the case, we have to apply the algorithm for this particular case. Otherwise, it behaves as described above.

Figure 7.3 illustrates a special kind of recursive association, where the two member ends of the *Association* are "the same" because one is the superclass of the other. In this case, the algorithm should also check whether the schema is satisfiable or not.

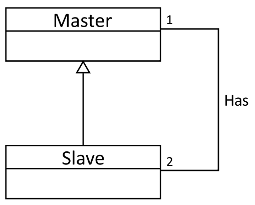


Figure 7.3: Example of a recursive relationship between a *general* and a *specific* class with non-satisfiable cardinality constraints.

In order to detect this new approach, we need to do some more tunings to our algorithm. First of all, step (3) has to be changed again: we have to check whether the two classes related by the *Association* are the same, or if one is more specific than the other.

Furthermore, we now have to take care of the *Generalizations*. When a *Generalization* is created, it may be the case that a binary association in the previous state between two classes becomes a recursive association, because one of them has become a subclass of the other. Similarly, when we remove a *Generalization*, the recursive association may become a regular binary association.

7.2.3 Results

Scenario 1

1. The modeler wants to create the schema defined in Fig. 3.6(a).

```
> new class
New Class Form
  Name: > Student
  Is it abstract? (y/N) >

> new class
```

```
New Class Form
  Name: > Curriculum
  Is it abstract? (y/N) >

> new assoc
New Binary Association Form
  Association name: > Enrolled
  First participant (classname): > Student
  First role name (student): >
  First min cardinality (0): > 20
  First max cardinality (1): > *
  Second participant (classname): > Curriculum
  Second role name (curriculum): >
  Second min cardinality (0): > 1
  Second max cardinality (1): > 1
```

2. The modeler wants to add the new association illustrated in Fig. 3.6(b).
3. The CMA detects the schema is unsatisfiable and defines the following task:
    - Unsatisfiable schema because of a conflict between associations “Likes(s: Student, likeCurriculum: Curriculum)” and “Enrolled(student: Student, curriculum: Curriculum)”.

```
> new assoc
New Binary Association Form
  Association name: > Likes
  First participant (classname): > Student
  First role name (student): > s
  First min cardinality (0): > 1
  First max cardinality (1): > 1
  Second participant (classname): > Curriculum
  Second role name (curriculum): > likedCurriculum
  Second min cardinality (0): > 1
  Second max cardinality (1): > 1

> ls tasks
tasks
  - [0] Unsatisfiable Schema because of a Conflict
        between Associations
        ‘Likes(s:Student, likedCurriculum:Curriculum)’ and
        ‘Enrolled(student:Student, curriculum:Curriculum)’
tasks> back
```

Scenario 2

1. The modeler wants the model defined in Fig. 7.2.
2. The CMA detects the schema is unsatisfiable and defines the following task:
    - Unsatisfiable schema because of Recursive Association “IsParentOf(parent: Person, child: Person)”.

```
> new class
New Class Form
  Name: > Person
  Is it abstract? (y/N) >
```

```
> new assoc
New Binary Association Form
  Association name: > IsParentOf
  First participant (classname): > Person
  First role name (person): > parent
  First min cardinality (0): > 2
  First max cardinality (1): > 2
  Second participant (classname): > Person
  Second role name (person): > child
  Second min cardinality (0): > 3
  Second max cardinality (1): > 3

> ls tasks
tasks
  - [0] Unsatisfiable Schema because of Recursive Association
        'IsParentOf(parent: Person, child: Person)'.
tasks> back
```

7.3 Schema Auto-Completion

In Sec. 3.3, we presented the Eclipse Platform as an example of an IDE. One of the features IDEs usually include is *code completion*. This feature is one of the most executed commands by developers, because it automatizes the work they have to do. The idea of this plugin is to follow a similar approach, helping modelers to complete their models automatically.

7.3.1 Description

In [11, 47], we can see the importance of using general ontologies while developing conceptual schemas. Creating conceptual schemas is difficult because it involves a modeler understanding a domain. However, by using ontologies the concepts can be automatically gathered and refined.

7.3.2 Implementation

This plugin does not use an ontology to gather new information, but uses hard-coded information to provide further assistance. The goal of this plugin is to demonstrate how a *CMA Command* can be included in the CMA and the effects of its execution.

The plugin only defines a new operation, called "Create a Class with Automatically Gathered Attributes". Consequently, we only need to define the `commands.xml` file and provide the *Command*'s name and the *CMA Command Effect*'s class name.

When the *Command* is executed, a *Class* name is queried to the modeler. If the provided name is one of the following, additional attributes are automatically introduced in the model:

Person has a *name*, a *surname*, an *age*, and an *identity document.*

Company has a *name* and *may be private.*

Product has a *name*, a *description*, a *code*, and *may have been certified by the European Union.*

Furthermore, the plugin also instantiates the following associations, if the related entity types are in the model:

WorksIn between a *Person* (*worker*) and a *Company.*

Owns between a *Person* (*owner*) and a *Company* (*property*).

Sells between a *Company* (*seller*) and a *Product*.

Despite the implementation is naive, it demonstrates how such a feature could improve modeler's work. A similar approach to the one presented in Sec. 7.1, when we connected to Google's API to perform spell checking, could be used: we may be able to connect to a external service which would provide some attribute or association suggestions based on the *Classes* defined in the model.

7.3.3 Results

1. The modeler wants to create the class "Person", and let the CMA populate the class with some attributes.

2. The modeler executes the following CMA operation:

   - Create a class with automatically gathered attributes and associations.

3. The CMA creates the class "Person" with the following attributes: "name", "surname", "age", and "identityDocument".

```
> cma
cma
  - [createcagaa] Create a Class with Automatically Gathered
                  Attributes and Associations

cma> createcagaa
  Class Name
  (Type '!cancel' at any time to quit this dialog.
    Class name: > Person

> ls classes
List of classes
  class Person
    attributes
      name: String -- [1]
      surname: String -- [1]
      age: UnlimitedNatural -- [1]
      identityDocument: String -- [1]
  end
```

4. The modeler wants to use the same operation to create a class named "Company".

5. The CMA creates the class "Company" with the following attributes: "name" and "isPrivate".

6. The CMA has also create the following associations between "Person" and "Company":

   - WorksIn(worker: Person, company: Company)
   - Owns(owner: Person, property: Company)

```
> cma createcagaa
  Class Name
  (Type '!cancel' at any time to quit this dialog.
    Class name: > Company

> ls classes
List of classes
  class Person
    attributes
      name: String -- [1]
      surname: String -- [1]
      age: UnlimitedNatural -- [1]
      identityDocument: String -- [1]
  end

  class Company
    attributes
      name: String -- [1]
      isPrivate: Boolean -- [1]
  end

> ls assocs
List of associations
  association WorksIn between
    Person[1..*] role worker
    Company[0..1] role company
  end

  association Owns between
    Person[1] role owner
    Company[0..1] role property
  end
```

7.4 Conclusions

As described in Ch. 5, the architecture is splitted in two levels to simplify the complexity of our CMA. The main benefit of this solution is that the addition of new features becomes quite simple, since a *method engineer* has to only populate the *knowledge level* in order to have new functionalities and criticisms running. Furthermore, the implementation we made aimed to maintain this simplicty by defining two components. The *plugin component* provides a mechanism to specifically populate the *knowledge level* seamlessly.

In this chapter we have seen the work a *method engineer* has to do is as difficult as the functionality he wants to implement. Note that this difficulty has nothing to do with the CMA, because what is important to us is if the *inclusion* of the functionality into the CMA is difficult, not the programming of the functionality itself. However, since our CMA does not constraint how a functionality has to be programmed, this programming can also be simplified. Take as an example the Google Spell Checker: despite programming an spell checker is quite complex, calling an external service is not.

8 Conclusions

In this chapter we present the main contributions of this master's thesis and some conclusions about the work done, and sketch a few ideas about future work that has to be done in order to have a fully functional CMA running on top of a real modeling CASE tool.

8.1 Master's Thesis Contributions

Despite the relevance conceptual modeling has in the software development process and, specially, the design of good conceptual schemas, we have seen current modeling tools do not provide enough assistance. Modeling tools are more focused on automating the development process by means of, for example, code generation, than on improving conceptual models itself. We have also seen that there are several tools that implement features which are of our interest. The main contribution of this master's thesis is the definition of an architecture for our CMA along with the implementation of a running prototype. The proposed architecture allows the integration in current modeling tools of these features.

Our approach involved adapting the CMA to a Platform Tool, which provided many advantages. First of all, it added an abstraction layer by which new features can be included in the Platform Tool. Hence, a *method engineer* could program a new feature for the CMA and all the Platform Tools to which the CMA is adapted would benefit. Furthermore, the architecture did not set any boundaries to how new features should be. Furthermore, our proposal entailed a flexibility that permited a *method engineer* to fully program new features inside the CMA, or to call external services instead. And last, but not least, the architecture was capable to efficiently manage *Tasks*. The usage of *Structural Events*, whose types were related to *Task Types*, allowed the CMA to control *Tasks*' behave efficiently, because for every single change in the model, only those potentially affected *Tasks* were considered.

In this thesis we also have shown an evaluation of the architecture by implementing a running prototype. On the one hand, we implemented the prototype itself. This prototype consisted of a Platform Tool which, in our case, was a custom modeling tool created for test purposes, and the CMA, which implemented the architecture we defined and which was adapted to the Platform Tool. By this implementation, we acquired useful information on how to deal with the adaptation of the CMA to a Platform Tool and how to create a plugin system to load new features, among others. On the other hand, in order to have a running prototype, we had to implement a few new features. We presented some examples of new features that could be added to our CMA. We showed the work a *method engineer* would have to do to extend the capabilities of a Platform Tool and we evaluated the complexity of this work.

8.2 Future Work

There are some open issues that should be addressed in order to complete the research in this work. First of all, the APIs we defined have to be extended. On the one hand, the UML API includes a simplified version of the UML metamodel, which we believe it is powerful enough to add interesting new features, since it includes the most important UML elements. Nonetheless, it is worth to extend it to cover more information. On the other hand, we defined an extremely simplified UI API. If we want a richer interaction with the modeler, more UI elements are required. We also believe that defining this richer UI screens declaratively (external files), instead of programmatically (inside the code), would simplify the generation and maintenance of plugins.

Second, another task is the adaptation of the CMA to a real modeling CASE tool. We have already demonstrated that the CMA can be adapted to a modeling tool, but using a real one would also be interesting. Furthermore, we could also evaluate the impact the adaptation of the CMA to a real modeling tool has. The CMA's goal is to evaluate the consequences of a *Command Effect*. Thus, every time a *Command Effect* modifies the model, the CMA analyzes the *Structural Events* generated and modifies the *Tasks* the modeler has to address. This additional work probably increases the response times of the modeling tool, and the additional information kept by the CMA increases the memory usage requirements of the modeling tool. We believe that a comparison of time and memory usage of a modeling Platform Tool with and without the CMA is important. Moreover, such an evaluation could be useful if different implementations of the CMA were provided, because they could be compared.

Finally, the CMA could be extended to include further customization to improve its behaviour. For example, in Ch. 7 we presented the following two plugins: a *naming* plugin which asked the modeler if a name is correct by showing her a sentence, and a *schema auto-completion* plugin which automatically includes attributes to the model. It is expected that the automatically gathered attributes follow the *naming* guidelines, but each time an attribute is automatically gathered and added to the model, the *Task* querying the modeler if the name is correct is created. An extension to the CMA could be to disable certain plugins when a certain *Command* is executed. Another example could be the inclusion of *repair plans* [12] to fix inconsistencies.

Bibliography

[1] ArgoUML. ArgoUML, `http://argouml.tigris.org`.

[2] Balzer, R. Tolerating inconsistency. In *ICSE '91: Proceedings of the 13th international conference on Software engineering* (1991), IEEE Computer Society Press, pp. 158–165.

[3] Beaton, W., and des Rivières, J. Eclipse platform: Technical overview. Tech. rep., The Eclipse Foundation, 2006.

[4] Berlage, T. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput.-Hum. Interact. 1*, 3 (1994), 269–294.

[5] Blanc, X., Mougenot, A., Mounier, I., and Mens, T. Incremental detection of model inconsistencies based on model operations. In *CAiSE '09: Proceedings of the 21st International Conference on Advanced Information Systems Engineering* (2009), Springer-Verlag, pp. 32–46.

[6] Boger, M., Sturm, T., and Fragemann, P. Refactoring browser for UML. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)* (2002), pp. 77–81.

[7] Cabot, J., Clarisó, R., and Riera, D. Verification of UML/OCL class diagrams using constraint programming. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop* (2008), IEEE Computer Society, pp. 73–80.

[8] Cabot, J., and Teniente, E. Determining the structural events that may violate an integrity constraint. In *in UML 2004, LNCS* (2004), Springer, pp. 173–187.

[9] Cadoli, M., Calvanese, D., and Mancini, T. Finite satisfiability of UML class diagrams by constraint programming. In *In Proc. of the 2004 International Workshop on Description Logics (DL'2004), volume 104 of CEUR Workshop Proceedings. CEUR-WS.org* (2004).

[10] Comai, S., and Tanca, L. Termination and confluence by rule prioritization. *IEEE Transactions on Knowledge and Data Engineering 15* (2003), 257–270.

[11] Conesa, J., de Palol, X., and Olivé, A. Building conceptual schemas by refining general ontologies. In *Database and Expert Systems Applications, 14th Int. Conf., Prague, Czech Republic* (2003), pp. 693–702.

[12] Da Silva, M. A. A., Mougenot, A., Blanc, X., and Bendraou, R. Towards automated inconsistency handling in design models. In *22nd International Conference on Advanced Information Systems Engineering, CAiSE 2010* (June 2010), no. 6051, Springer Lecture Notes in Computer Science (LNCS), pp. 348–362.

[13] Dubinsky, Y., Humayoun, S. R., and Catarci, T. Eclipse plug-in to manage user centered design. In *Proceedings of the First Workshop on the Interplay between Usability Evaluation and Software Development* (2008).

[14] Eclipse Community. MDT-UML2Tools, `http://wiki.eclipse.org/MDT-UML2Tools`.

[15] Egyed, A. Instant consistency checking for the UML. In *ICSE '06: Proceedings of the 28th international conference on Software engineering* (2006), ACM, pp. 381–390.

[16] Ernst, J. What is metamodeling?, `http://infogrid.org/wiki/Reference/WhatIsMetaModeling`.

[17] Farré, C., Teniente, E., and Urpí, T. Checking query containment with the CQC method. *Data Knowl. Eng. 53*, 2 (2005), 163–223.

[18] Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996.

[19] Fowler, M., Beck, K., Brant, J., Opdyke, W. F., and Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[20] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[21] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming* (1993), Springer-Verlag, pp. 406–431.

[22] Gane, C. *Computer-Aided Software Engineering: the Methodologies, the Products, and the Future*. Prentice-Hall, Inc., 1990.

[23] Geer, D. Eclipse becomes the dominant Java IDE. *Computer 38* (2005), 16–18.

[24] Gentleware. Poseidon for UML, `http://www.gentleware.com/`.

[25] Gogolla, M., Büttner, F., and Richters, M. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming 69*, 1-3 (2007), 27–34.

[26] Hall, B. H., and Khan, B. Adoption of new technology. Tech. Rep. 1055, 2003.

[27] Halpin, T., and Curland, M. Automated verbalization for ORM 2. In *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops* (2006), pp. 1181–1190.

[28] Huang, R. Making active CASE tools—toward the next generation CASE tools. *SIGSOFT Softw. Eng. Notes 23*, 1 (1998), 47–50.

[29] IBM. Rational Rose modeler, `http://www-01.ibm.com/software/awdtools/developer/rose/modeler/`.

[30] Ideogramic. Ideogramic UML, `http://www.ideogramic.com/products/uml/product-info.html`.

[31] Jarzabek, S., and Huang, R. The case for user-centered CASE tools. *Commun. ACM 41*, 8 (1998), 93–99.

[32] Kalyanpur, A., Halaschek-Wiener, C., Kolovski, V., and Hendler, J. Effective NL paraphrasing of ontologies on the semantic web.

[33] Kelly, S., Lyytinen, K., and Rossi, M. Metaedit+: A fully configurable multi-user and multi-tool CASE and CAME environment. In *CAiSE '96: Proceedings of the 8th International Conference on Advances Information System Engineering* (1996), Springer-Verlag, pp. 1–21.

[34] Kop, C. Towards a combination of three representation techniques for conceptual data modeling. *Advances in Databases, First International Conference on 0* (2009), 95–100.

[35] Layman, L. M., Williams, L. A., and St. Amant, R. Mimec: Intelligent user notification of faults in the Eclipse IDE. In *CHASE '08: Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering* (2008), ACM, pp. 73–76.

[36] Magicdraw. Magic Draw, `http://www.magicdraw.com`.

[37] Meziane, F., Athanasakis, N., and Ananiadou, S. Generating natural language specifications from UML class diagrams. *Requir. Eng. 13*, 1 (2008), 1–18.

[38] Neustupny, T., Thompson, D., and Morris, T. ArgoUML users wiki: Hot topics, `http://www.argouml-users.net/index.php?title=Hot_Topics`.

[39] Object Management Group (OMG). *Unified Modeling Language (UML), Superstructure – version 2.2*, 2009.

[40] Olivé, A. *Conceptual Modeling of Information Systems.* Springer, 2007.

[41] Queralt, A., and Teniente, E. Reasoning on UML conceptual schemas with operations. In *Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings* (2009), pp. 47–62.

[42] Ramírez, A., Vanpeperstraete, P., Rueckert, A., Odutola, K., Bennett, J., Tolke, L., and van der Wulp, M. ArgoUML user manual: A tutorial and reference description. Tech. rep., 2000–2009.

[43] Robbes, R., and Lanza, M. How program history can improve code completion. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (2008), IEEE Computer Society, pp. 317–326.

[44] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

[45] Selic, B. The pragmatics of model-driven development. *Software, IEEE 20*, 5 (2003), 19–25.

[46] Spanoudakis, G., and Zisman, A. Inconsistency management in software engineering: Survey and open research issues. In *In Handbook Of Software Engineering and Knowledge Engineering* (2001), World Scientific, pp. 329–380.

[47] Sugumaran, V., and Storey, V. C. The role of domain ontologies in database design: an ontology management and conceptual modeling environment. *ACM Trans. Database Syst. 31*, 3 (2006), 1064–1094.

[48] Takeda, H., Veerkamp, P., Tomiyama, T., and Yoshikawa, H. Modeling design processes. *AI Mag. 11*, 4 (1990), 37–48.

[49] Tarling, B. Issue 1934: Implement undo, `http://argouml.tigris.org/issues/show_bug.cgi?id=1834`.

[50] Towner, L. E. *CASE: Concepts and Implementation.* McGraw-Hill, Inc., 1989.

[51] Vaishnavi, V., and Kuechler, W. Desing research in information systems, 2004.

[52] Vessey, I., Jarvenpaa, S. L., and Tractinsky, N. Evaluation of vendor products: CASE tools as methodology companions. *Commun. ACM 35*, 4 (1992), 90–105.

[53] Villegas, A., and Olivé, A. On computing the importance of entity types in large conceptual schemas. *Advances in Conceptual Modeling-Challenging Perspectives* (2009), 22–32.

[54] Villegas, A., and Olivé, A. A method for filtering large conceptual schemas. In *Conceptual Modeling - ER 2010* (2010), (To Appear).

[55] Visual Paradigm. Visual Paradigm for UML, `http://www.visual-paradigm.com/product/vpuml/`.

[56] Wand, Y., and Weber, R. Research commentary: Information systems and conceptual modeling – a research agenda. *Info. Sys. Research 13*, 4 (2002), 363–376.

A Review of the Used Patterns

This appendix reviews the primary patterns used throughout this master thesis. First, the *Accountability Pattern* is presented. This analysis pattern introduces the concepts of *knowledge level* and *operational level* to deal with the complexity we may found while modeling hierarchical structures.

Then, a wide range of design patterns are reviewed. We first describe the *Abstract Factory Pattern*, which provides an interface for creating families of related objects. Subsequently, we describe the *Adapter Pattern*, which converts the interface of a class into another one. We then see the *Command Pattern*, which encapsulates operations into regular classes and, thus, it simplifies the inclusion of an undo/redo functionality in an application. Next, we present the *Publish/Subscribe Pattern*, also known as *Observer Pattern*, which defines how to detect when an object changes so that all its dependents are notified. Finally, we describe the *Proxy Pattern*, which provides a surrogate for another object to control access to it or to perform additional actions whenever the original object is accessed.

A.1 Analysis Patterns

In [18], Fowler defines a *pattern* as "an idea that has been useful in one practical context and will probably be useful in others". Analysis patterns reflect conceptual structures, which are untied from actual software implementations.

The only analysis pattern we see is the *Accountability Pattern*, because it introduces the concept of a *knowledge level* when the complexity and variability of what we are working with is high.

A.1.1 Accountability Pattern

Throughout chapter 2 of [18], Fowler presents the concept of accountability, and proposes a pattern to deal with it. Accountability "applies when a person or organization is responsible to another".

Fowler uses the organization structure problem to show the development of the accountability model. As he states in sections 2.2–2.5, it is quite common for companies to be organized into different levels, such as operating units, which are divided into regions, which are divided into divisions, which are divided into sales offices, etc. He questions how to model this domain and proposes different solutions, discussing the pros and cons of each one.

Basically, he concludes that it is better to model this structures using types and associations, constrained by rules, between those types than using a taxonomy of classes. He argues that "it is easier to change a rule than to change the model structure". As complexity is introduced, the rules for defining types become more and more complex. This complexity can be managed by introducing a *knowledge level*. Using a knowledge level splits the model into two sections: the *operational* and the *knowledge* levels.

At the operational level, the model records the day to day events of the domain. At the knowledge level, the model records the general rules that govern this structure. Note that the operational level is always related to the knowledge level, because its types and, thus, its behaviour is defined there.

A.2 Design Patterns

As Gamma et al. state in [21], design patterns identify, name, and abstract common themes in object-oriented design. Design patterns have many uses in the object-oriented development process:

- They provide a common vocabulary, defining abstractions that raise the level at which one programs, so, at the end, the system complexity is reduced.
- They constitute a reusable base of experience and knowledge.
- They reduce the learning time of new libraries and provide a target for the refactoring of class hierarchies.

They consist of three essential parts:

- An abstract description of a set of classes and its structure.
- The issue in system design addressed by the abstract structure.
- The consequences of applying the abstract structure to a system's architecture.

The design patterns presented in the following subsections have been studied in depth by Gamma et al. in [20]. The figures used to describe each pattern are based on those they use in their work.

A.2.1 Abstract Factory Pattern

The *Abstract Factory Pattern* is a *creational pattern*, which means that it abstracts the instantiation process. More specifically, this pattern "provides an interface for creating families of related or dependent objects without specifying their concrete classes".

A typical example of this pattern is a user interface toolkit that supports multiple look-and-feel standards. Different look-and-feels define different appearances for "widgets", like scroll bars or buttons. Depending on the platform on top of which the application is running, it may instantiate one widget or another.

In Figure A.1, we see the main components of this pattern. There is an interface[1] named *AbstractFactory* that defines the callable operations. These operations are supposed to create new *AbstractProducts*. The types they return are, in turn, interfaces. A concrete implementation of the *AbstractFactory* returns concrete implementations of the *AbstractProducts*.

Some consequences of using this pattern are:

[1]We do not care whether it is an interface or an abstract class.

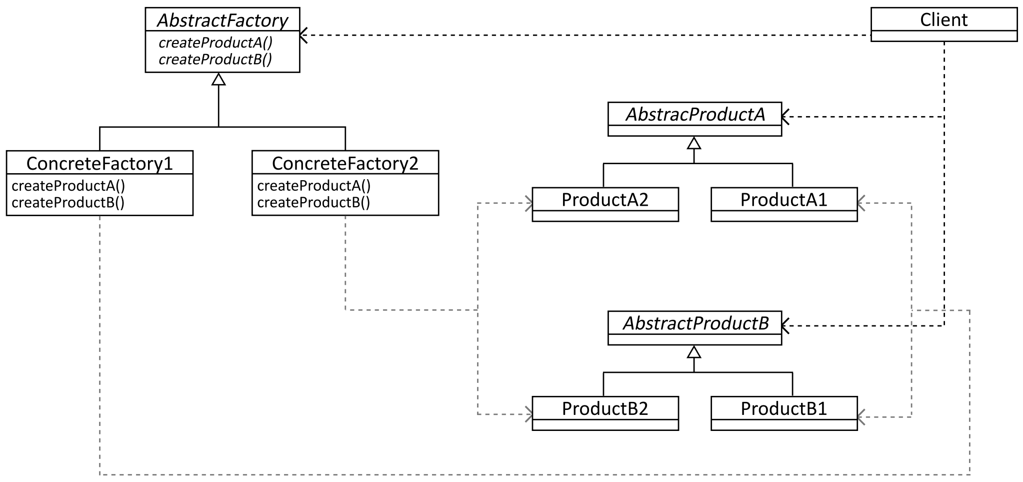


Figure A.1: Structure of the Adapter Pattern.

- It isolates concrete classes.
- It makes exchanging product families easy, because it only requires to change the concrete factory.
- Supporting new kinds of products is difficult, because the *AbstractFactory* interfaces fixes the set of products that can be created.

A.2.2 Adapter Pattern

The *Adapter Pattern* is a *structural pattern.* Its intent is "to convert the interface of a class into another interface clients expect. Thus, classes that are incompatible because of its interfaces can work together".

Consider for example a drawing editor that lets users arrange graphical elements into pictures. The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. Certain *Shapes* are easy to implement, such as *LineShapes* or *Polygon-Shapes*, but it is really complex to implement a *TextShape*, so the programmer might want to use a toolkit that already implements a *TextView.*

In order to include this *TextView*, which probably has a different and incompatible interface, the programmer could change the *TextView* class so it conforms to the *Shape* interface. This solution requires access to the toolkit's source code, which is not always available, and modify it, which is difficult and not desirable, because one should be able to adapt it without modifying the original code. A better solution would be to define a *TextShape* that *adapts* the *TextView* interface to *Shape*'s.

Adapting the *TextView* to the *Shape*'s interface can be achieved in one of two ways: (1) by inheriting *Shape*'s interface and *TextView*'s implementation or (2) by compositing a *TextView* instance within a *TextShape*, and implementing *TextShape* in terms of *TextView*'s interface.

Figure A.2 shows the adapter pattern using object composition. The *Target* interface define the set of operations available to the *Client*, and the *Adapter* adapts an *Adaptee* so the later conforms the *Target* interface.

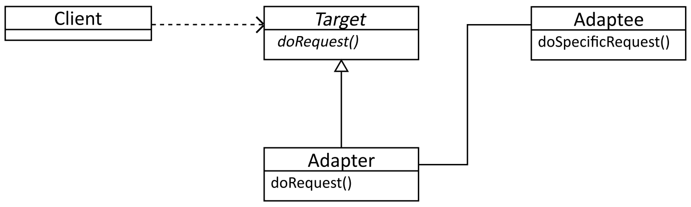


Figure A.2: Structure of the Adapter Pattern.

A.2.3 Command Pattern

The *Command Pattern* is a *behavioural pattern* in which an object encapsulates all the information needed to call a method at a later time.

Sometimes it is necessary to issue requests to objects without knowing anything about the operation being requested. A typical example is a user interface toolkit, which includes buttons and menus that carry out a request in response to user input. The toolkit can not implement the request explicitly, because only applications using the toolkit know what should be done on which object.

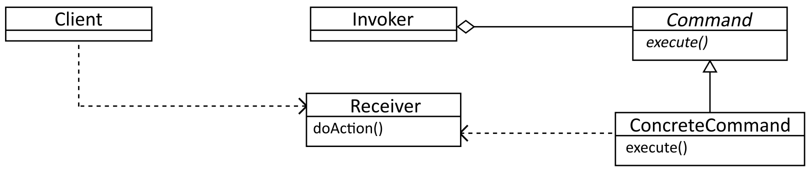


Figure A.3: Structure of the Command Pattern.

In Figure A.3 we see that the abstract *Command* declares an interface for executing operations. A *ConcreteCommand* stores the *Reciever* as an instance variable and implements the *execute* method to invoke the request. The *Receiver* has the knowledge required to carry out the request.

Some consequences of using this pattern are:

- *Command* decouples the object that invokes the operation from the one that knows how to perform it.
- *Commands* can be manipulated and extended like any other object.
- *Commands* can be assembled into a "composite command".
- It is easy to add new *Commands*.

The Undo/Redo Functionality

One of the main advantages of using *Commands* to implement operations is that the effects of these operations are properly encapsulated and perfectly defined inside a *Command* object. This feature implies that, as long as a *Command* knows how to perform an action, it also knows how to undone the effects of this action.

Furthermore, we have seen that *Commands* can be manipulated like any other object. This means that we can create a data structure to store as many issued *Commands* as required, and then move backwards and towards this list of *Commands* to undo and redo, respectively, their effects.

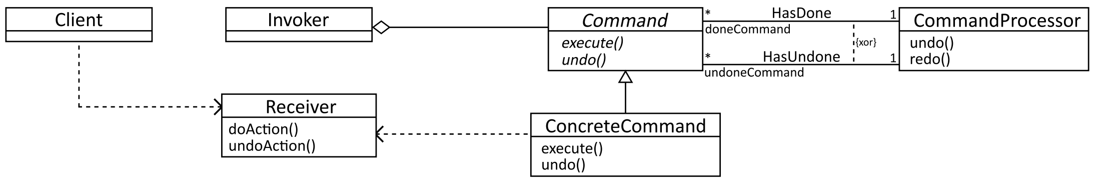


Figure A.4: Structure of the Command Pattern with Undo/Redo Capabilities.

Figure A.4 shows the changes required to include the undo/redo capabilities to this pattern. Basically, we add a *CommandProcessor* element which stores all the issued *Commands* via the associations *HasDone* and *HasUndone*, and an *undo* method to undo the changes introduced by the *Command*.

Usually, the concrete performance of this functionality is the following:

1. Every time the user issues a *Command*, it is executed and stored in the *HasDone* association.

2. When the user decides to undo the effect of the last *Command*, the *Command* is retrieved from the *HasDone* association and its *undo* method is called. Once the operation is undone, it is removed from the *HasDone* association and it is stored in the *HasUndone* association.

3. If the user rollback to undo an *undo*, the *CommandProcessor* acts in a similar manner: it retrieves the last undone *Command* from the *HasUndone* association and executes its *execute* method. Finally, it removes it from the *HasUndone* association and stores it in the *HasDone* association.

4. Generally, whenever a new *Command* is issued, the *HasUndone* association is reset.

A.2.4 Publish/Subscribe –Observer– Pattern

The *Publish/Subscribe Pattern*, also known as *Observer Pattern*, is a *behavioural pattern.* It defines "a one-to-many dependency between objects so that when one object changes, all its dependents are notified and updated automatically".

Partitioning a system into a collection of cooperating classes requires some further work to maintain consistency between the related objects. For example, a spreadsheet application that can show data in a chart should automatically update the chart whenever the data is modified. This behaviour implies that the spreadsheet and the chart are dependent.

The *Observer Pattern* describes how to establish these relationships using *subjects* and *observers.* A *subject* may have any number of *observers* that require to be notified whenever the *subject* changes its state.

Figure A.5 shows the main elements found in this pattern. We can easily see that there is a *Subject* which can have as many *Observers* as required. Whenever we want to make an object dependent to another one, the former has to subclass the *Observer* and the later has to subclass the *Subject.*

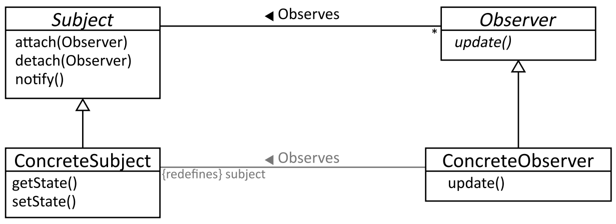


Figure A.5: Structure of the Publish/Subscribe (Observer) Pattern.

A.2.5 Proxy Pattern

The *Proxy Pattern* is a *structural pattern*. It "provides a surrogate or placeholder for another object to control access to it". This pattern is specially useful whenever we want to reference another object in a more complex manner. There are several situations in which the *Proxy Pattern* is applicable:

- A *virtual proxy* creates expensive objects on demand.
- A *protection proxy* controls access to the original object.
- A *smart reference* is a replacement for a bare pointer that performs additional actions whenever an object is accessed.
- etc.

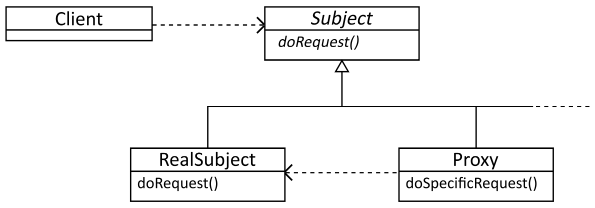


Figure A.6: Structure of the Proxy Pattern.

In Figure A.6 we can see the structure of the *Proxy Pattern*. The *Proxy* maintains a reference that permits the access the real subject. Note that both the *Proxy* and the *RealSubject* subclass *Subject*, which means the *Proxy* can be substituted for the real subject.

B UML API Documentation

B.1 UMLFactory

```
public interface UMLFactory
```

The *Factory* for the model. It provides a create method for each non-abstract class of the model.

createBinaryAssociation

```
public Association createBinaryAssociation(Type src, Type dest) throws
    OperationExecutionFailedException
```

Returns a new object of class `Association`. This instance has two *Member Ends*, which are those specified in the creation statement.

Parameters

src The `Type` of the first participant.

dest The `Type` of the second participant.

Returns

A new object of class `Association`.

Exceptions

OperationExecutionFailedException

createClass

```
public Class createClass(String name) throws
    OperationExecutionFailedException
```

Returns a new object of class `Class`, named *name*.

Parameters

name The name of the new `Class`.

Returns

A new object of class `Class`.

Exceptions

OperationExecutionFailedException

createDataType

```
public DataType createDataType(String name) throws
    OperationExecutionFailedException
```

Returns a new object of class `DataType`, named *name*.

Parameters

name The name of the new `DataType`.

Returns

A new object of class `DataType`.

Exceptions

OperationExecutionFailedException

createEnumeration

```
public Enumeration createEnumeration(String name) throws
    OperationExecutionFailedException
```

Returns a new object of class `Enumeration`, named *name*.

Parameters

name The name of the new `Enumeration`.

Returns

A new object of class `Enumeration`.

createGeneralization

```
public Generalization createGeneralization(Classifier general,
    Classifier specific) throws OperationExecutionFailedException
```

Returns a new object of class `Generalization`.

Parameters

general The value of the *General* reference.

specific The value of the *Specific* reference.

Returns

A new object of class `Generalization`.

Exceptions

OperationExecutionFailedException

createPrimitiveType

```
public PrimitiveType createPrimitiveType(String name) throws
    OperationExecutionFailedException
```

Returns a new object of class `PrimitiveType`, named *name*.

Parameters

name The name of the new `PrimitiveType`.

Returns

A new object of class `PrimitiveType`.

Exceptions

OperationExecutionFailedException

B.2 UMLUtilities

```
public interface UMLUtilities
```

This interface defines some useful functions to perform common tasks. It may be used to retrieve information about the current working UML model.

Any class implementing this interface must implement the *Singleton* pattern, because there is a *getInstance()* method.

getAllAssociations

```
public List<Association> getAllAssociations()
```

Returns all the instances of associations of `Association` found in the schema.

Returns

All the instances of associations of `Association` found in the schema.

getAllClasses

```
public List<Class> getAllClasses()
```

Returns all the instances of associations of `Class` found in the schema.

Returns

All the instances of associations of `Class` found in the schema.

getAllDataTypes

```
public List<DataType> getAllDataTypes()
```

Returns all the instances of associations of `DataType` found in the schema.

Returns

All the instances of associations of `DataType` found in the schema.

getAllEnumerations

```
public List<Enumeration> getAllEnumerations()
```

Returns all the instances of associations of `Enumeration` found in the schema.

Returns

All the instances of associations of `Enumeration` found in the schema.

getAllPrimitiveTypes

```
public List<PrimitiveType> getAllPrimitiveTypes()
```

Returns all the instances of associations of `PrimitiveType` found in the schema.

Returns

All the instances of associations of `PrimitiveType` found in the schema.

getAssociation

```
public Association getAssociation(String name) throws
    AssociationNotFoundException
```

Returns the `Association` whose name is *name*.

Parameters

name The *name* of the `Association`.

Returns

The `Association` whose name is *name*.

Exceptions

AssociationNotFoundException

getClazz

```
public Class getClazz(String name) throws ClassNotFoundException
```

Returns the `Class` whose name is *name*.

Parameters

name The *name* of the `Class`.

Returns

The `Class` whose name is *name*.

Exceptions

ClassNotFoundException

getDataType

```
public DataType getDataType(String name) throws
    DataTypeNotFoundException
```

Returns the `DataType` whose name is *name*.

Parameters

name The *name* of the `DataType`.

Returns

The `DataType` whose name is *name*.

Exceptions

DataTypeNotFoundException

getEnumeration

```
public Enumeration getEnumeration(String name) throws
    EnumerationNotFoundException
```

Returns the `Enumeration` whose name is *name*.

Parameters

name The *name* of the `Enumeration`.

Returns

The `Enumeration` whose name is *name*.

Exceptions

EnumerationNotFoundException

getPrimitiveType

```
public PrimitiveType getPrimitiveType(String name) throws
    PrimitiveTypeNotFoundException
```

Returns the `PrimitiveType` whose name is *name*.

Parameters

name The *name* of the `PrimitiveType`.

Returns

The `PrimitiveType` whose name is *name*.

Exceptions

PrimitiveTypeNotFoundException

getElement

```
public Element getElement(ElementIdentifier id) throws
    ElementNotFoundException
```

Returns the `Element` whose identifier is *id*.

Parameters

id The *id* of the `Element`.

Returns

The `Element` whose identifier is *id*.

Exceptions

ElementNotFoundException

B.3 Association

```
public interface Association extends Classifier
```

A representation of the model object *Association.* An association describes a set of tuples whose values refer to typed instances. An instance of an association is called a link.

getMemberEnd

```
public Property getMemberEnd(String name) throws
    MemberEndNotFoundException
```

Retrieves `Property` with the specified *Name* from the *Member End* reference list.

Parameters

name The *Name* of the `Property` to retrieve

Returns

The `Property` with the specified *Name.*

Exceptions

MemberEndNotFoundException

getMemberEnds

```
public List<Property> getMemberEnds()
```

Returns the value of the *Member End* reference list. The list contents are of type `Property`.

Returns

The value of the *Member End* reference list.

destroy

```
@Override public void destroy() throws OperationExecutionFailedException
```

Destroys this element by removing all cross references to/from it and removing it from its containing resource or object.

Exceptions

OperationExecutionFailedException

B.4 Class

```
public interface Class extends Classifier
```

A representation of the model object *Class*. A class describes a set of objects that share the same specifications of features, constraints, and semantics.

createOwnedAttribute

```
public Property createOwnedAttribute(String name, Type type) throws
    OperationExecutionFailedException
```

Creates a new `Property`, with the specified *Name*, and *Type*, and appends it to the *Owned Attribute* containment reference list.

Parameters

name The *Name* for the new `Property`.

type The *Type* for the new `Property`

Returns

The new `Property`.

Exceptions

OperationExecutionFailedException

getOwnedAttribute

```
public Property getOwnedAttribute(String name) throws
    OwnedAttributeNotFoundException
```

Retrieves the first `Property` with the specified *Name* from the *Owned Attribute* containment reference list.

Parameters

name The *Name* for the new `Property`.

Returns

The `Property` with the specified Name.

Exceptions

OwnedAttributeNotFoundException

getOwnedAttributes

```
public List<Property> getOwnedAttributes()
```

Returns the value of the *Owned Attribute* containment reference list. The list contents are of type `Property`.

Returns

The value of the *Owned Attribute* containment reference list.

getSuperClasses

```
public List<Class> getSuperClasses()
```

Returns the value of the *Super Class* reference list. The list contents are of type `Class`.

Returns

The value of the *Super Class* reference list.

B.5 Classifier

```
public interface Classifier extends Type
```

A representation of the model object *Classifier*. A classifier is a classification of instances - it describes a set of instances that have features in common.

B.6 DataType

```
public interface DataType extends Classifier
```

A representation of the model object *Data Type*. A data type is a type whose instances are identified only by their value. A data type may contain attributes to support the modeling of structured data types.

createOwnedAttribute

```
public Property createOwnedAttribute(String name, Type type) throws
    OperationExecutionFailedException
```

Creates a new `Property`, with the specified *Name*, and *Type*, and appends it to the *Owned Attribute* containment reference list.

Parameters

name The *Name* for the new `Property`.

type The *Type* for the new `Property`

Returns

The new `Property`.

Exceptions

OperationExecutionFailedException

getOwnedAttribute

```
public Property getOwnedAttribute(String name) throws
    OwnedAttributeNotFoundException
```

Retrieves the first `Property` with the specified *Name* from the *Owned Attribute* containment reference list.

Parameters

name The *Name* for the new `Property`.

Returns

The `Property` with the specified *Name*.

Exceptions

OwnedAttributeNotFoundException

getOwnedAttributes

```
public List<Property> getOwnedAttributes()
```

Returns the value of the *Owned Attribute* containment reference list. The list contents are of type `Property`.

Returns

The value of the *Owned Attribute* containment reference list.

B.7 Element

```
public interface Element
```

A representation of the model object *Element*. An element is a constituent of a model. As such, it has the capability of owning other elements.

destroy

```
public void destroy() throws OperationExecutionFailedException
```

Destroys this element by removing all cross references to/from it and removing it from its containing resource or object.

Exceptions

OperationExecutionFailedException

equals

```
public boolean equals(Element element)
```

Indicates whether the other `Element` is *equal to* this one. Two elements are *equal* if they both have the same `ElementIdentifier`.

Parameters

element the other `Element` with which to compare.

Returns

whether the other `Element` is *equal to* this one. Two elements are *equal* if they both have the same `ElementIdentifier`.

getIdentifier

```
public ElementIdentifier getIdentifier()
```

Returns the `ElementIdentifier` that uniquely identifies this `Element`.

Returns

the `ElementIdentifier` that uniquely identifies this `Element`.

B.8 Enumeration

```
public interface Enumeration extends DataType
```

A representation of the model object *Enumeration*. An enumeration is a data type whose values are enumerated in the model as enumeration literals.

createOwnedLiteral

```
public EnumerationLiteral createOwnedLiteral(String name) throws
    OperationExecutionFailedException
```

Creates a new `EnumerationLiteral`, with the specified *Name*, and appends it to the *Owned Literal* containment reference list.

Parameters

name The *Name* for the new `EnumerationLiteral`.

Returns

The new `EnumerationLiteral`.

Exceptions

OperationExecutionFailedException

getOwnedLiteral

```
public EnumerationLiteral getOwnedLiteral(String name) throws
    EnumerationLiteralNotFoundException
```

Retrieves the first `EnumerationLiteral` with the specified *Name* from the *Owned Literal* containment reference list.

Parameters

name The *Name* for the new `EnumerationLiteral`.

Returns

The `EnumerationLiteral` with the specified *Name*.

Exceptions

EnumerationLiteralNotFoundException

B.9 EnumerationLiteral

```
public interface EnumerationLiteral extends NamedElement
```

A representation of the model object *Enumeration Literal*. An enumeration literal is a user-defined data value for an enumeration.

getEnumeration

```
public Enumeration getEnumeration()
```

Returns the value of the `Enumeration` container reference.

Returns

The value of the `Enumeration` container reference.

B.10 Generalization

```
public interface Generalization extends Element
```

A representation of the model object *Generalization*. A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.

A generalization relates a specific classifier to a more general classifier, and is owned by the specific classifier.

getGeneral

```
public Classifier getGeneral()
```

Returns the value of the *General* reference.

Returns

The value of the *General* reference.

getSpecific

```
public Classifier getSpecific()
```

Returns the value of the *Specific* container reference.

Returns

The value of the *Specific* container reference.

B.11 MultiplicityElement

```
public interface MultiplicityElement extends Element
```

A representation of the model object *Multiplicity Element.* A multiplicity is a definition of an inclusive interval of non-negative integers beginning with a lower bound and ending with a (possibly infinite) upper bound. A multiplicity element embeds this information to specify the allowable cardinalities for an instantiation of this element.

getLower

```
public int getLower()
```

Returns the value of the *Lower* attribute.

Returns

The value of the *Lower* attribute.

getUpper

```
public int getUpper()
```

Returns the value of the *Upper* attribute.

Returns

The value of the *Upper* attribute.

setLower

```
public void setLower(int lower)
```

Sets the value of the *Lower* attribute.

Parameters

lower the new value of the *Lower* attribute.

setUpper

```
public void setUpper(int upper)
```

Sets the value of the *Upper* attribute. Internally, an upper value set to -1 means *.

Parameters

upper the new value of the *Upper* attribute.

B.12 NamedElement

```
public interface NamedElement extends Element
```

A representation of the model object *Named Element.* A named element is an element in a model that may have a name. A named element supports using a string expression to specify its name.

getName

```
public String getName()
```

Returns the value of the *Name* attribute. The name of the *NamedElement.*

Returns

The value of the *Name* attribute.

setName

```
public void setName(String name) throws
    OperationExecutionFailedException
```

Sets the value of the *Name* attribute.

Parameters

name The new value of the *Name* attribute.

Exceptions

OperationExecutionFailedException

B.13 PrimitiveType

```
public interface PrimitiveType extends DataType
```

A representation of the model object *Primitive Type.* A primitive type defines a predefined data type, without any relevant substructure (i.e., it has no parts in the context of UML). A primitive datatype may have an algebra and operations defined outside of UML, for example, mathematically.

B.14 Property

```
public interface Property extends MultiplicityElement, NamedElement
```

A representation of the model object *Property*. A property is a structural feature of a classifier that characterizes instances of the classifier.

A property related by *ownedAttribute* to a classifier (other than an association) represents an attribute and might also represent an association end. It relates an instance of the class to a value or set of values of the type of the attribute.

A property related by memberEnd or its specializations to an association represents an end of the association. The type of the property is the type of the end of the association.

When a property is an attribute of a classifier, the value or values are related to the instance of the classifier by being held in slots of the instance. When a property is an association end, the value or values are related to the instance or instances at the other end(s) of the association

The range of valid values represented by the property can be controlled by setting the property's type.

getAssociation

```
public Association getAssociation()
```

Returns the value of the `Association` reference. References the association of which this property is a member, if any.

Returns

The value of the `Association` reference.

getClazz

```
public Class getClazz()
```

Returns the value of the `Class` reference.

Returns

The value of the `Class` reference.

getDataType

```
public DataType getDataType()
```

Returns the value of the `DataType` container reference.

Returns

The value of the `DataType` container reference.

getType

```
public Type getType()
```

Returns the value of the `Type` reference. The type of the Property.

Returns

The value of the `Type` reference.

setType

```
public void setType(Type type)
```

Sets the value of the `Type` reference.

Parameters

type the new value of the `Type` reference.

B.15 Type

```
public interface Type extends NamedElement
```

A representation of the model object *Type*. A type is a named element that is used as the type for a typed element. A type can be contained in a package. A type constrains the values represented by a typed element.

C UI API Documentation

C.1 UIFactory

```
public interface UIFactory
```

The *Factory* for the UI widgets. It provides a create method for `Menu`s, `QuestionDialog`s and `ModalWindows`.

createMenu

```
public Menu createMenu(String name)
```

Returns a new object of class `Menu`, named *name*.

Parameters

name The name of the new `Menu`.

Returns

A new object of class `Menu`.

createQuestionDialog

```
public QuestionDialog createQuestionDialog(String name, String question)
```

Returns a new object of class `QuestionDialog`, named *name* and querying the user whether he accepts or not the *question*.

Parameters

name The name of the new `QuestionDialog`.

question The question the user has to accept or decline.

Returns

A new object of class `QuestionDialog`.

createModalWindow

```
public ModalWindow createModalWindow(String name)
```

Returns a new object of class `ModalWindow`, named *name*.

Parameters

name The name of the new `ModalWindow`.

Returns

A new object of class `ModalWindow`.

C.2 UINamedElement

```
public String getName()
```

Returns the name of this `UINamedElement`.

Returns

The name of this `UINamedElement`.

setName

```
public void setName(String name)
```

Sets the value of the *name* attribute.

Parameters

name The new value of the *name* attribute.

C.3 InputText

```
public interface InputText extends UINamedElement
```

A representation of an input text UI element. This widget provides an interface to retrieve information from the user. The widget has a *label* and the field where the user is supposed to write in.

getValue

```
public String getValue()
```

Returns the value the user introduced.

Returns

the value the user introduced.

C.4 Item

```
public interface Item extends Node
```

An *Item* is a special type of `Node`. Unlike a `Menu`, it cannot contain children. Thus, it is a leaf in a `Menus` hierarchy.

Any leaf node in a `Menu` hierarchy has an associated action. Such action is a `CMACommand`, and it is executed when its *Item* containment is selected.

C.5 Menu

```
public interface Menu extends Node
```

A *Menu* is a special type of `Node` that may contain additional nodes inside. Those nodes can be either a *Menu* or an `Item`.

createSubMenu

```
public Menu createSubMenu(String name)
```

Creates a new `Menu` named *name* inside this `Menu`.

Parameters

name the name of this new sub-menu.

Returns

@returns the new sub-menu.

createItem

```
public Item createItem(String name, CMACommand command)
```

Creates a new `Menu` named *name*inside this `Menu`. When this item is selected, the `CMACommand` is executed.

Parameters

name the name of this new item.

command the `CMACommand` that must be executed when this item is selected.

Returns

@returns the new item inside this `Menu`.

C.6 ModalWindow

```
public interface ModalWindow extends UINamedElement
```

A representation of a window designed to elicit a response from the user. This version of the API, which is a prototype, only allows the creation of `InputText` as widgets intended to retrieve information from the user.

createInputText

```
public InputText createInputText(String name)
```

Returns a new object of class `InputText`, named *name*.

Parameters

name The name of the new `InputText`.

Returns

A new object of class `InputText`.

show

```
public void show()
```

Shows the `ModalWindow` to the user, so she can interact with it.

isDataValid

```
public boolean isDataValid()
```

Returns false if the user closes the `ModalWindow`, or true otherwise.

Returns

false if the user closes the `ModalWindow`, or true otherwise.

C.7 Node

```
public interface Node extends UINamedElement
```

A *Node* is the superclass of `Menu` and `Item` classes. It defines the name of a node inside a `Menu`, regardless its concrete type.

C.8 QuestionDialog

```
public interface QuestionDialog extends UINamedElement
```

A representation of an independent subwindow meant to query something to the user. It show a question which has to be accepted or decline by the user.

show

```
public void show()
```

Shows the `ModalWindow` to the user, so he can interact with it.

acceptedByUser

```
public boolean acceptedByUser()
```

Returns true if the user says *Yes* to the question, false otherwise.

Returns

true if the user says *Yes* to the question, false otherwise.

D Plugin XML Files

This appendix shows the XML files associated to the plugins presented in Ch. 7.

D.1 Naming

D.1.1 commands.xml

```
<?xml version="1.0" encoding="utf-8"?>
<commands>
   <command id="0" corrector="true">
      <effect classname="lettercase.effects.
          CapitalizeFirstLetterClassOrAssociationName" />
         <name>
           Capitalize the First Letter of the Name Automatically
         </name>
   </command>

   <command id="1" corrector="true">
      <effect classname="lettercase.effects.
          UncapitalizeFirstLetterPropertyName" />
         <name>
            Change the First Letter of the Name to Lower Case
            Automatically
         </name>
   </command>

   <command id="2" corrector="true">
      <effect classname="spelling.effects.FixNameSpellingEffect" />
         <name>
            Provide a Correct Spell Suggestion (via Google)
         </name>
   </command>

   <command id="3" corrector="true">
      <effect classname="guidelines.effects.
          NamingGuidelinesCheckerEffect" />
         <name>
            Check if the NamedElement Follows the Naming Guidelines.
         </name>
   </command>
```

```
</commands>
```

D.1.2 tasks.xml

```
<?xml version="1.0" encoding="utf-8"?>
<tasks>
   <!-- A NamedElement has to have a name... -->
   <task classname="guidelines.tasks.NamingGuidelinesTask">
      <name>
         There is a NamedElement that has not been given a name.
      </name>
      <description>
         The name of a named element is optional. It is important to
         name all the elements of a conceptual schema properly. This
         task outlines that there is an element that has no name yet.

         Note that there are some special cases where a named element
         does not require a name. This is the case of a Property that
         is a memberEnd of an association. Its name is supposed to be
         the name of its type, starting with a non capital letter.
      </description>

      <structural-events>
         <create element="NamedElement">
            <generation condition="
                DefaultTaskGeneratorForClassInstanceCreation" />
         </create>

         <delete element="NamedElement">
            <cancellation condition="
                DefaultTaskFinalizerForClassInstanceDeletion" />
         </delete>

         <set default="true" element="NamedElement" attribute="name">
            <cancellation condition="
                DefaultTaskFinalizerForAttributeSetter" />
         </set>

         <set default="false" element="NamedElement" attribute="name">
            <cancellation condition="
                DefaultTaskFinalizerForAttributeSetter" />
         </set>

      </structural-events>
   </task>

   <!-- Classes, Associations and Enumerations must start with a capital
        letter -->
   <task classname="lettercase.tasks.
       InvalidCapitalizationClassOrAssociationNameTask">
      <name>
         Invalid capitalization of ‘#namedelement#’.
      </name>
      <description>
         Normally, classes, associations, and enumerations begin with a
         capital letter. The name ‘#namedelement#’ is unconventional
         because it does not begin with a capital.

         Following good naming conventions help to improve the
         understandability and maintainability of the design.
      </description>

      <correctors>
         <corrector command="0" />
      </correctors>

      <structural-events>
         <set default="false" element="Class" attribute="name">
            <generation condition="lettercase.conditions.
                UncapitalizedClassOrAssociationNameRetriever" />
            <achievement condition="lettercase.conditions.
                CapitalizedClassOrAssociationNameChecker" />
         </set>

         <set default="true" element="Class" attribute="name">
            <generation condition="lettercase.conditions.
                UncapitalizedClassOrAssociationNameRetriever" />
            <achievement condition="lettercase.conditions.
                CapitalizedClassOrAssociationNameChecker" />
         </set>

         <delete element="Class">
            <cancellation condition="
                DefaultTaskFinalizerForClassInstanceDeletion" />
         </delete>

         <set default="false" element="Association" attribute="name">
            <generation condition="lettercase.conditions.
                UncapitalizedClassOrAssociationNameRetriever" />
            <achievement condition="lettercase.conditions.
                CapitalizedClassOrAssociationNameChecker" />
         </set>

         <set default="true" element="Association" attribute="name">
            <generation condition="lettercase.conditions.
                UncapitalizedClassOrAssociationNameRetriever" />
            <achievement condition="lettercase.conditions.
                CapitalizedClassOrAssociationNameChecker" />
         </set>

         <delete element="Association">
            <cancellation condition="
                DefaultTaskFinalizerForClassInstanceDeletion" />
         </delete>

         <set default="false" element="Enumeration" attribute="name">
            <generation condition="lettercase.conditions.
                UncapitalizedClassOrAssociationNameRetriever" />
            <achievement condition="lettercase.conditions.
                CapitalizedClassOrAssociationNameChecker" />
         </set>

         <set default="true" element="Enumeration" attribute="name">
            <generation condition="lettercase.conditions.
                UncapitalizedClassOrAssociationNameRetriever" />
            <achievement condition="lettercase.conditions.
                CapitalizedClassOrAssociationNameChecker" />
         </set>

         <delete element="Enumeration">
```

```
            <cancellation condition="
                DefaultTaskFinalizerForClassInstanceDeletion" />
         </delete>

      </structural-events>
   </task>

   <!-- Properties must start with a low-case letter -->
   <task classname="lettercase.tasks.
       InvalidCapitalizationPropertyNameTask">
      <name>
         Invalid capitalization of '#propertyname#'.
      </name>
      <description>
         Normally, properties, like an attribute or an association's
         member end, begin with a lowercase letter. The name
         '#propertyname#' is unconventional because it does not.

         Following good naming conventions help to improve the
         understandability and maintainability of the design.
      </description>

      <correctors>
         <corrector command="1" />
      </correctors>

      <structural-events>
         <set default="false" element="Property" attribute="name">
            <generation condition="lettercase.conditions.
                CapitalizedPropertyNameRetriever" />
            <achievement condition="lettercase.conditions.
                UncapitalizedPropertyNameChecker" />
         </set>

         <set default="true" element="Property" attribute="name">
            <generation condition="lettercase.conditions.
                CapitalizedPropertyNameRetriever" />
            <achievement condition="lettercase.conditions.
                UncapitalizedPropertyNameChecker" />
         </set>

         <delete element="Property">
            <cancellation condition="
                DefaultTaskFinalizerForClassInstanceDeletion" />
         </delete>

      </structural-events>
   </task>

   <!--
      Naming Guidelines: names must follow the naming guidelines we
          defined.
      This means that the presented sentence have to make sense.
   -->
   <task classname="guidelines.tasks.NamingGuidelinesTask">
      <name>
         Does '#namedelement#' Follow the Naming Guidelines?
      </name>

      <correctors>
         <corrector command="3" />
      </correctors>
      <structural-events>
         <set default="false" element="NamedElement" attribute="name">
            <generation condition="
                DefaultTaskGeneratorForAttributeSetter" />
         </set>

         <delete element="NamedElement">
            <cancellation condition="
                DefaultTaskFinalizerForClassInstanceDeletion" />
         </delete>

      </structural-events>
   </task>

   <!-- Default names must be overwritten by names provided by the
       modeler. -->
   <task classname="guidelines.tasks.NamingGuidelinesTask">
      <name>
         The NamedElement ‘#namedelement#’ was given a default name.
      </name>
      <description>
         Some tools provide default names for the NamedElements of
         the domain. The modeler should know that these names were
         not given by him, but by the tool.

         This task reminds the modeler to check if default names
         are OK, or should be changed instead.
      </description>

      <structural-events>
         <set default="true" element="NamedElement" attribute="name">
            <generation condition="
                DefaultTaskGeneratorForAttributeSetter" />
         </set>

         <set default="false" element="NamedElement" attribute="name">
            <cancellation condition="
                DefaultTaskFinalizerForAttributeSetter" />
         </set>

         <delete element="NamedElement">
            <cancellation condition="
                DefaultTaskFinalizerForClassInstanceDeletion" />
         </delete>

      </structural-events>
   </task>

   <!--
      This task checks if a name was properly spelled, using Google’s
          functionality
      ‘‘Did you mean: ...’’
   -->
   <task classname="spelling.tasks.MisspelledNameTask">
      <name>
         Is ‘#name#’ properly spelled?
      </name>
      <description>
         Google’s spell checking software, used by this task, checks
         whether your name uses the most common spelling of a given
         word. If it thinks you’re likely to be wrong, it provides a
```

```
         different spelling.
      </description>

      <correctors>
         <corrector command="2" />
      </correctors>

      <structural-events>
         <set default="false" element="NamedElement" attribute="name">
            <generation condition="spelling.conditions.
                MisspelledNameChecker" />
            <achievement condition="spelling.conditions.
                ProperlySpelledNameChecker" />
         </set>

         <set default="true" element="NamedElement" attribute="name">
            <generation condition="spelling.conditions.
                MisspelledNameChecker" />
            <achievement condition="spelling.conditions.
                ProperlySpelledNameChecker" />
         </set>

         <delete element="NamedElement">
            <cancellation condition="
                DefaultTaskFinalizerForClassInstanceDeletion" />
         </delete>

      </structural-events>
   </task>

</tasks>
```

D.2 Schema Satisfiability

D.2.1 tasks.xml

```
<?xml version="1.0" encoding="utf-8"?>
<tasks>
   <task classname="tasks.UnsatisfiableSchemaTask">
      <name>
         Unsatisfiable Schema because of #reason#
      </name>
      <description>
         In general, conceptual schemas include many integrity
         constraints.  A schema ‘S’ is satisfiable if it admits
         at least one legal instance of an information base.
         For some constraints, it may happen that only empty
         or nonfinite information bases satisfy them. In
         conceptual modeling, the information bases of interest
         are finite and may be populated.

         This method checks whether a schema is strongly
         satisfiable using the two cardinality constraints
         defined for binary relationship types.

         If the schema is unsatisfiable, check the cardinalities
         of the related associations.
      </description>
      <structural-events>
         <set default="false" element="Property" attribute="upper">
            <generation condition="conditions.
                InvalidMemberEndMultiplicitiesRetriever" />
            <achievement condition="conditions.
                ValidMemberEndMultiplicitiesChecker" />
         </set>
         <set default="false" element="Property" attribute="lower">
            <generation condition="conditions.
                InvalidMemberEndMultiplicitiesRetriever" />
            <achievement condition="conditions.
                ValidMemberEndMultiplicitiesChecker" />
         </set>

         <set default="true" element="Property" attribute="upper">
            <generation condition="conditions.
                InvalidMemberEndMultiplicitiesRetriever" />
            <achievement condition="conditions.
                ValidMemberEndMultiplicitiesChecker" />
         </set>
         <set default="true" element="Property" attribute="lower">
            <generation condition="conditions.
                InvalidMemberEndMultiplicitiesRetriever" />
            <achievement condition="conditions.
                ValidMemberEndMultiplicitiesChecker" />
         </set>

         <delete element="Association">
            <cancellation condition="conditions.
                UnsatisfiableAssociationDeletedChecker" />
         </delete>

         <create element="Generalization">
            <generation condition="conditions.
                InvalidMemberEndMultiplicitiesGeneralizationRetriever"
                />
         </create>

         <unlink
           first-role="generalization" first-element="Generalization"
           second-role="general" second-element="Class">
            <cancellation condition="conditions.
                UnsatisfiableSchemaGenerationDeletedChecker" />
         </unlink>

      </structural-events>
   </task>

</tasks>
```

D.3 Schema Auto-Completion

D.3.1 commands.xml

```
<?xml version="1.0" encoding="utf-8"?>
<commands>
   <command id="0">
      <effect classname="effects.
          ClassWithAutomaticAttributesGatheringCreationEffect" />
```

```
         <name>
            Create a Class with Automatically Gathered Attributes
            and Associations
         </name>
   </command>

</commands>
```