UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

The UML Modeling Tool of ProjectIT-Studio

João Paulo Pedro Mendes de Sousa Saraiva
(Licenciado)

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Orientador: Doutor Alberto Manuel Rodrigues da Silva

DOCUMENTO PROVISÓRIO

Dezembro 2006

Abstract

ProjectIT is a collaborative research project, developed in the context of the Information Systems Group of INESC-ID, that integrates several final graduation works, MSc and PhD thesis. Its main objective is to provide a complete software development workbench, with support for activities such as project management, requirements engineering, analysis, design, and code generation. This objective is achieved through the implementation of a CASE tool, called ProjectIT-Studio, that covers all the stages of an IT product's life-cycle.

The context of this work is ProjectIT-MDD, a functional component of ProjectIT that allows the visual specification of models and the automatic generation of artifacts from these models. After the previous work (which resulted in "Eclipse.NET, an integration platform for ProjectIT-Studio"), this work focuses on the creation of an UML 2.0 visual modeling plugin for ProjectIT-Studio, fully integrated with the requirements and code-generation capabilities of ProjectIT-Studio, that allows the seamless transition from requirements specification to source-code generation according to current MDE approaches.

The main goals of this work are: (1) the development of an UML graphical modeling plugin for ProjectIT-Studio that provides features of a flexible UML modeling tool; (2) the complete integration of this plugin with the other plugins of ProjectIT-Studio; (3) the support of the UML Profile mechanism, in a way that is both simple and powerful; (4) the development of a mechanism that constantly monitors the syntactic and semantic consistency of a model, according to the UML Profiles applied to it; and (5) the preliminary development of a mechanism that allows the application of user-defined model-to-model transformations. Thus, the results of this work will allow ProjectIT-Studio users to quickly create visual models of information systems, and then supply these models as input to an automatic code generator.

Keywords

ProjectIT, ProjectIT-Studio, MDE, UML, Information Systems, modeling, integration.

Resumo

O ProjectIT é um projecto de investigação, desenvolvido no contexto do Grupo de Sistemas de Informação do INESC-ID, que integra vários Trabalhos Finais de Curso e teses de Mestrado e Doutoramento. O seu objectivo principal é fornecer uma plataforma completa de desenvolvimento de *software*, com suporte para actividades como gestão de projectos, engenharia de requisitos, análise, desenho, e geração de código. Este objectivo é atingido através da implementação de uma ferramenta CASE, chamada ProjectIT-Studio, que cobre todas as etapas do ciclo de vida de um produto de TI.

O contexto deste trabalho é o ProjectIT-MDD, um componente funcional do ProjectIT que permite a especificação visual de modelos e a geração automática de artefactos a partir desses modelos. Após o trabalho anterior (que originou o Eclipse.NET, uma plataforma de integração para o ProjectIT-Studio), este trabalho foca-se na criação de um módulo de modelação visual em UML para o ProjectIT-Studio, totalmente integrado com as restantes funcionalidades do ProjectIT-Studio, que permite a transição da especificação de requisitos para a geração de código de acordo com as abordagens MDE actuais.

Os objectivos principais deste trabalho são: (1) o desenvolvimento de um módulo de modelação visual em UML para o ProjectIT-Studio, que inclua funcionalidades típicas de ferramentas de modelação UML; (2) a integração deste módulo com os outros módulos do ProjectIT-Studio; (3) o suporte para o mecanismo de Perfis UML, de um modo simples e poderoso; (4) o desenvolvimento de um mecanismo que monitorize a consistência sintáctica e semântica de um modelo, de acordo com os Perfis UML que lhe tenham sido aplicados; e (5) o desenvolvimento preliminar de um mecanismo de aplicação de transformações modelo-para-modelo definidas pelo utilizador. Os resultados deste trabalho irão permitir que os utilizadores do ProjectIT-Studio criem rapidamente modelos visuais de sistemas de informação, e de seguida forneçam esses modelos a um gerador automático de código.

Palavras-chave

ProjectIT, ProjectIT-Studio, MDE, UML, Sistemas de Informação, modelação, integração.

Acknowledgements

I would like to demonstrate my gratitude to all my teachers, who provided me with the knowledge and know-how necessary to overcome any challenges and learn from them.

I would especially like to thank Professor Alberto Silva (who supervised me throughout this work), for the opportunity he gave me to do this work, for his constant availability and excellent guidance, for the enthusiasm he managed to transmit to me and to the ProjectIT team, for motivating me throughout the duration of this work, and for all the career opportunities he gave me.

Besides the role assumed by teachers in the learning process, I believe it's important to recognize the importance of the work environment and collaboration of fellow colleagues in knowledge-sharing and as an engineering skills enhancement driver. Therefore, I would like to show my gratitude to my ProjectIT colleagues and friends, David Ferreira and Rui Silva, for their support and friendship over this last year.

The outside of the work environment is just as important, and I could not end this section without mentioning my closest friends (in no special order): João Pombinho, João Gonçalves, Rui Eugénio, Telmo Nabais, Ricardo Clérigo, Carlos Santos and Frederico Baptista (or, as a colleague once labeled us, "The Galactics"). To all of them, I now express my gratitude; they were always there for me, even though I wasn't.

Finally, I couldn't have reached this stage in my life if it wasn't for my parents, Carlos Alberto de Sousa Saraiva and Maria Isabel Pedro Mendes de Sousa Saraiva, who always supported me and provided me with all they could. No words could even begin to express how much I truly owe them; thus, I can only try to express my gratitude, and my regret for the little availability and little time devoted to them over these last two years of hard work.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The development of software information systems is a very complex activity, involving non-trivial issues in various areas such as technology and human resources. Despite the efforts made to overcome the issues related with this activity, these issues are still frequently found in software development projects, with its negative consequent aspects in terms of project scope, time, budget and quality. One of the main causes for this situation is the fact that many projects do not follow a structured, standard and systematic approach, like the methodologies and best practices proposed by the Software Engineering community [Silva 01].

Facing this situation, the Information Systems Group of INESC-ID [INESC-ID] started an initiative in the area of requirements engineering, project management, and model-driven development, named **ProjectIT** [Silva 04]. This initiative's main goal is to minimize the negative consequences mentioned above, by researching new approaches to accelerate the underlying development process by automating its repetitive activities (which are error-prone and introduce unnecessary complexity in the final product) thus globally improving the quality of the development process. These new approaches are being implemented in two prototypes (illustrated in Figure 1.1): **ProjectIT-Studio** and **ProjectIT-Enterprise**. The idea of these prototypes is to provide a CASE (Computer-Assisted Software Engineering) tool, which should keep in mind Software Engineering's best practices and simultaneously be methodology-independent. Figure 1.2 presents the components of ProjectIT's functional architecture.

So far, the main results of this project are: (1) a new requirements specification language, called ProjectIT-RSL [Videira 05]; (2) an Unified Modeling Language (UML) [OMG 05b] profile called "eXtreme modeling Interactive Systems" (XIS), which was proposed and validated in previous work [Silva 03a, Silva 03b] and is currently in its second version; and (3) a set of integrated tools, called ProjectIT-Studio [Saraiva 05a, Saraiva 05b, Silva 06a, Ferreira 06, Silva 06b], that covers the activities of the entire soft-

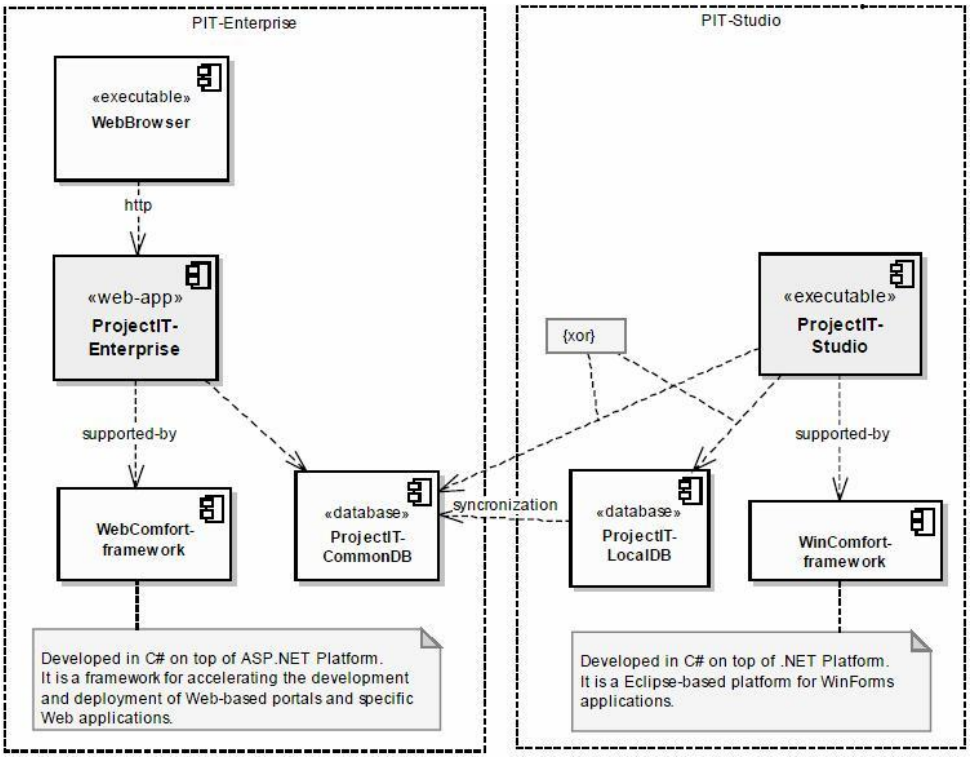

Figure 1.1: ProjectIT applicational architecture (extracted from [Silva 05b]).

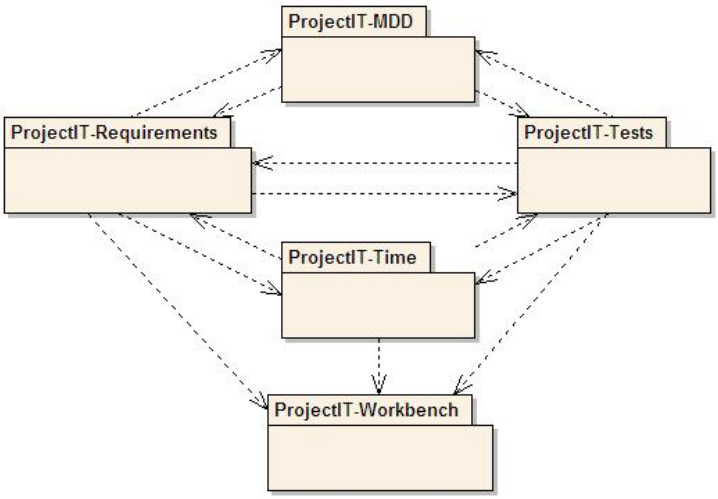

Figure 1.2: ProjectIT functional architecture (extracted from [Silva 06a]).

ware development life cycle, from requirements specification to the generation of artifacts (such as source code and documentation) through the application of generative programming techniques.

Interactive systems are a sub-class of information systems that provide a large number of common features and functionalities, such as user-interfaces to drive the human-machine interaction, databases to keep information consistent, and role-based user and

access control management [Silva 07]. ProjectIT promotes a platform-independent approach, based on Model-Driven Architecture (MDA) [MDA], for designing interactive systems: (1) specify and validate requirements by using ProjectIT-RSL; (2) obtain a high-level UML model corresponding to the specified requirements; (3) refine the obtained model by using the XIS profile, which allows the design of interactive systems at a PIM ("Platform-Independent Model") level, according to MDA terminology; and finally (4) generate artifacts by using model-to-code transformations specific to different deployment platforms, such as Web, desktop or mobile specific platforms. Figure 1.3 illustrates the **ProjectIT approach** with the concrete application of the ProjectIT-RSL language and the XIS profile.

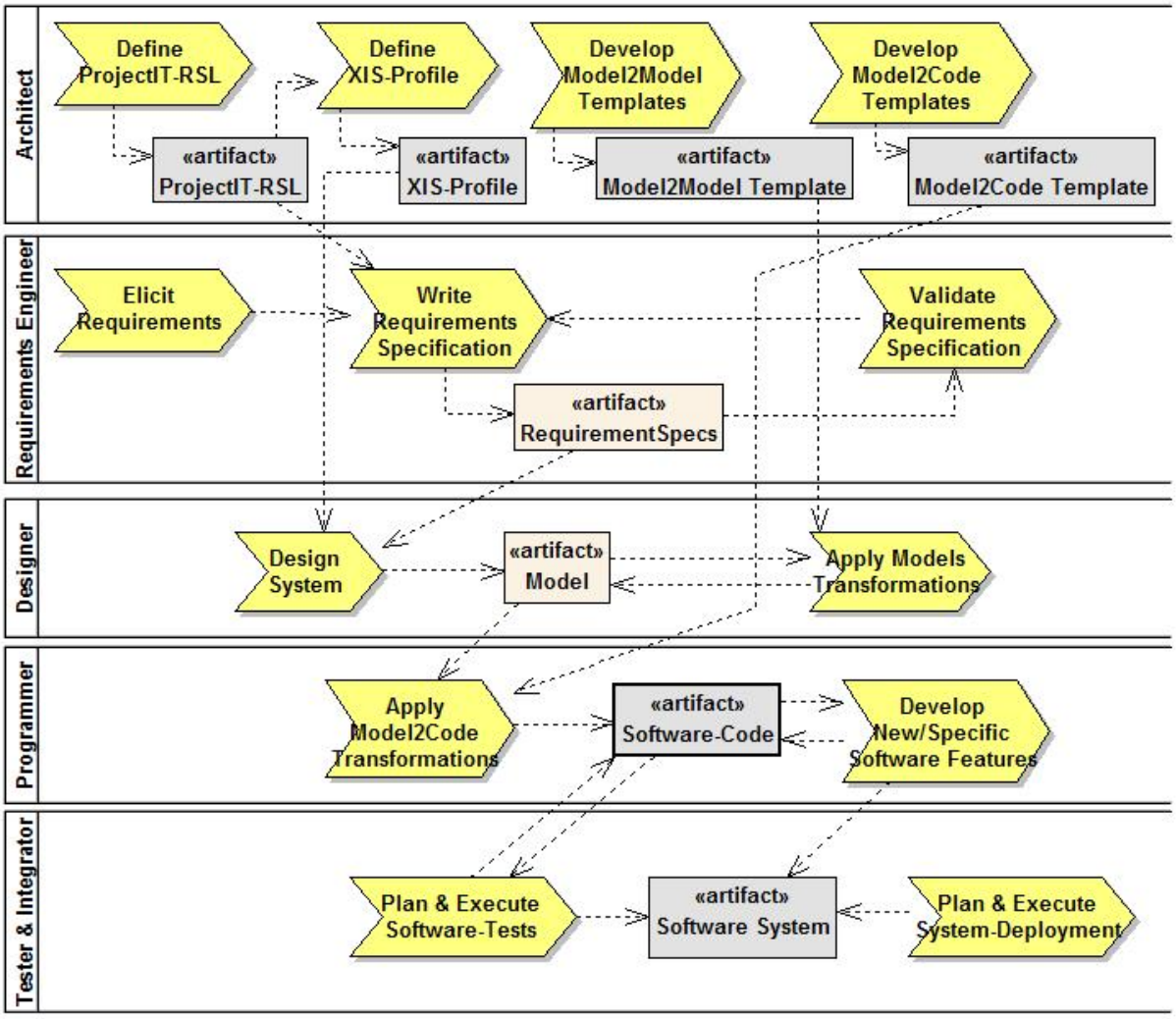

Figure 1.3: Roles and processes involved in the ProjectIT approach's workflow (extracted from [Silva 06a]).

The ProjectIT approach is language-independent, which means the Designer is free to choose any UML profile that is considered appropriate to design the system. However, in the current research, using the XIS profile is recommended.

1.1 Motivation

As suggested in Figure 1.3, UML model design is considered a fundamental principle of the ProjectIT approach. This requires that ProjectIT-Studio provide an UML modeling plugin which not only accelerates the traditional task of completely modeling a system, but also features a high level of integration with other ProjectIT-Studio plugins, such as the ProjectIT-Studio/Requirements [Ferreira 06] and the ProjectIT-Studio/MDDGenerator [Silva 06b].

Besides this integration, the plugin must also provide an efficient extensibility mechanism. Considering that the ProjectIT approach is language-independent, the plugin must allow the Architect to specify the semantics that are inherent to a specific UML profile. Additionally, the plugin must also allow the specification of possible model transformations, which are a key factor of the ProjectIT approach.

This work describes the creation of **ProjectIT-Studio/UMLModeler**, a modeling plugin for ProjectIt-Studio that fulfills these requirements and is primarily focused on making the user look upon the ProjectIT approach as an easy and efficient way to model interactive systems.

1.2 Context

This work takes place in the context of **ProjectIT-MDD**, which is ProjectIT's functional component related to the area of information systems' modeling and Model-Driven Engineering (MDE) [Schmidt 06]. This component allows the specification of models, transformations between models defined in different languages, and the automatic generation of artifacts (such as source-code and documentation). Figure 1.4 presents a high-level overview of this component's architecture; the shaded packages indicate the sub-components of ProjectIT-MDD which were developed in the context of this work.

This work was developed in the context of the Information Systems Group of INESC-ID, in strict collaboration with the rest of the ProjectIT-Studio development team, and was supervised by PhD Alberto Manuel Rodrigues da Silva.

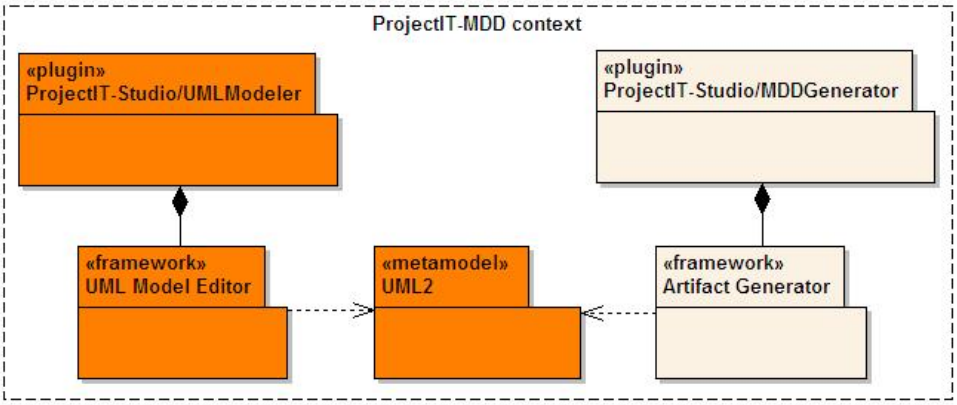

Figure 1.4: ProjectIT-MDD's architecture (extracted from [Silva 06a]).

1.3 Objectives

The main objectives of this work are: (1) the development of a graphical modeling plugin for ProjectIT-Studio, called **ProjectIT-Studio/UMLModeler**, that provides features typical of traditional UML modeling tools; (2) the complete integration of this plugin with the other plugins of ProjectIT-Studio; (3) the support of the UML Profile mechanism, in an elegant and powerful way; (4) the development of a mechanism that constantly monitors the syntactic and semantic consistency of a model, according to the profiles applied to it; and (5) the development of a mechanism that allows the application of user-defined model-to-model transformations.

From the user's perspective, the modeling plugin should facilitate the realization of the most common tasks in the ProjectIT approach (such as stereotype application and tagged-value edition), so that the user can model systems in a quick and easy way.

This work was validated with several case studies, featuring various levels of complexity. This validation was conducted by performing the following steps: (1) define the requirements for the desired system (using ProjectIT-Studio/Requirements); (2) model the system using ProjectIT-Studio/UMLModeler; and finally (3) generate the implementation of the system using ProjectIT-Studio/MDDGenerator.

It is expected that the results of this work will allow ProjectIT-Studio users to easily adopt and follow the ProjectIT approach, by being able to quickly create visual models of information systems and then supply these models as input to ProjectIT-Studio/MDDGenerator, as is suggested in Figure 1.3.

1.4 Document Outline

This MSc dissertation is organized around seven chapters and three appendices, whose contents are described in the next paragraphs.

Chapter 1 – Introduction. This chapter provides a brief overview of the ProjectIT research program, the ProjectIT approach and the prototype tool that supports it: ProjectIT-Studio. Moreover, the context and main objectives of this thesis are also described.

Chapter 2 – State of the Art. This chapter describes the concepts, techniques, tools, and frameworks that were analyzed in the context of this work. This analysis is particularly centered around: (1) MDE, (2) existing UML modeling tools and their features, and (3) technological frameworks that were either used in this work, or provided some fundamental ideas about what would be the most important factors to consider during the development of ProjectIT-Studio/UMLModeler.

Chapter 3 – ProjectIT-Studio Context. This chapter presents ProjectIT-Studio in greater detail, namely its high-level component architecture. Additionally, the various individual plugins that constitute ProjectIT-Studio are also presented, although not in great detail (since they were developed in parallel with this work).

Chapter 4 – The UML Modeling Plugin in ProjectIT-Studio. This chapter presents a thorough description of the ProjectIT-Studio/UMLModeler modeling plugin. The various internal components of the plugin are presented, as well as the technological and architectural decisions that were taken during the course of its development.

Chapter 5 – Supporting the XIS2 UML Profile. This chapter describes how the XIS2 profile was defined in ProjectIT-Studio/UMLModeler, as a way to validate the plugin as an effective way to specify and apply UML profiles.

Chapter 6 – Conclusions and Future Work. The conclusions drawn from all the developed work are presented, as well as the future work that should be performed in order to significantly enhance its performance and its user-interaction experience.

Bibliography. Lists the bibliographical references that were found relevant either for research purposes or for the development of the modeling plugin itself.

Glossary. This appendix presents some keywords mentioned in this document (such as acronyms and technical terms), describing their meaning.

Appendix A – The "MyOrders2" Case Study. This appendix presents one of the most complex case studies that was used in the validation of ProjectIT-Studio's plugins, and the results achieved.

Appendix B – ProjectIT-Studio Designer's Manual. This appendix consists of an in-depth guide describing how to use ProjectIT-Studio/UMLModeler. All the features offered by the plugin are also explained in detail.

Appendix C – ProjectIT-Studio Programmer's Manual. This appendix describes the internal mechanisms of ProjectIT-Studio/UMLModeler, and how developers can add new features to this plugin.

1.5 Conventions Used

This thesis is written using the following conventions: (1) the **bold face** font is used to highlight important concepts, tools and technology that are being presented for the first time in this thesis; (2) the *italic* font is used to emphasize a segment of text; (3) the `type writer` font is used for programming language keywords or code segments (e.g., names of classes, attributes, methods, or namespaces) developed or used in this work; and (4) any text enclosed in quotation marks (") and followed by a reference means that the text was extracted from the referenced document.

Whenever a figure is extracted or adapted from another document, the figure's caption will explicitly indicate this fact and make a reference to the corresponding document. All the original figures created for this thesis (i.e., not extracted or adapted from some external source) have no such reference. When suitable, figures will consist of diagrams specified using the UML language.

Chapter 2

State of the Art

This chapter presents the concepts, tools, and supporting technologies that are relevant to the work of this thesis. It is divided in three sections: (1) a presentation of the Model-Driven Engineering (MDE) paradigm and some current MDE-related standards, such as the Unified Modeling Language (UML); (2) a comparative analysis of some popular UML modeling tools; and (3) a presentation of the technological frameworks considered important for the implementation of ProjectIT-Studio/UMLModeler.

2.1 Model-Driven Engineering (MDE)

Model-Driven Engineering (MDE) is an emerging technique based on the systematic use of models as first-class entities during the solution specification [ModelWare, Schmidt 06]. The MDE paradigm treats the software development process as a set of transformations between models, from requirements to deployment, passing by analysis, design, and implementation. The idea behind MDE is that the real-world problem can be represented by a set of models that capture the essence of the solution's expected behavior. Unlike previous software development paradigms based on source code as a first-class entity, in this approach models become the first-class entities; source code, documentation, and other development artifacts can then be obtained from those models (much like compilers, which translate high-level language programs into their equivalent machine-language implementations).

There are already multiple initiatives, languages and approaches related to MDE, such as the Unified Modeling Language (UML) [UML], the MetaObject Facility (MOF) [MOF], the Model-Driven Architecture (MDA) [MDA] or the Domain-Specific Modeling (DSM) [DSMForum, Kelly 05]. Usually, MDE approaches are a combination of the usage of certain languages and a method for obtaining artifacts (source code and documentation) from models specified by using those languages. The main difference between the existing

MDE approaches is the offered software development life-cycle stages' coverage and how each of these stages are handled by the approach (e.g., if the models obtained during that stage are used for code generation, or transformed into other models).

The Object Management Group (OMG) [OMG] is a consortium that produces and promotes computer industry specifications. Originally aimed at setting standards for distributed object-oriented systems, it is now focused on modeling (of programs, systems, and business processes) and model-based standards. The OMG is responsible for some well-known modeling-related standards, such as the Unified Modeling Language (UML) [UML], the MetaObject Facility (MOF), the XML Metadata Interchange (XMI) [XMI] format, and the Query/Views/Transformations (QVT) [OMG 05a]. OMG also created its own approach to MDE, the Model-Driven Architecture (MDA) [MDA], which is based on UML, MOF, XMI, and QVT.

The MDE approach followed by the ProjectIT-MDD component is primarily based on the UML 2.0 and the MDA standards, which are presented in the following subsections.

2.1.1 Unified Modeling Language (UML) 2.0

The **Unified Modeling Language (UML)** [UML] is a general-purpose modeling language, originally designed to specify, visualize, construct, and document information systems. Nevertheless, it is not restricted to modeling software, being commonly used for business process modeling or representing organizational structures, for example.

UML was fundamental in the beginning of MDE, as it established consensus on the various graphical shapes used to represent concepts like classes, inheritance, states, etc. This was the first step in establishing a "universal" software modeling language, thus allowing developers to effectively transmit their own ideas using a standard notation. UML 2.0 represented a significant leap over its previous version, UML 1.5; although the most significant changes are about the language's internal architecture (i.e., the way how the UML 2.0 metamodel is defined), there were also changes that affect users, such as new diagram types (UML 2.0 has 13 diagram types versus the 9 diagram types of previous versions), new concepts and notations [Silva 05a, OMG 05b].

The UML 2.0 standard is organized into four different parts [France 06]: (1) the Infrastructure [OMG 06c], which provides the basic modeling constructs used by both UML 2.0 and the MetaObject Facility (MOF) 2.0, which is the metamodel used to specify UML; (2) the Superstructure [OMG 05b], which is the UML metamodel itself; (3) the Object Constraint Language (OCL) [OMG 06a], which provides a language for specifying queries, constraints, and additional operations in UML models (the "additional operations" specified in the UML Superstructure specification are written in OCL, as Listing

2.1 illustrates), although these operations do not change the state of the model; and (4) the Diagram Interchange (DI) [OMG 06b], which is an extension to the UML metamodel that supports the exchange of UML diagram layouts for interoperability between UML modeling tools. Figure 2.1 illustrates the relationship between the UML Infrastructure, the UML Superstructure (i.e., the UML metamodel) and the MOF.

Listing 2.1: Example of an operation expressed in OCL (extracted from [OMG 05b]).

```
NamedElement::allNamespaces(): Sequence(Namespace);
allNamespaces =
    if self.namespace->isEmpty()
        then Sequence{}
    else
        self.namespace.allNamespaces()->prepend(self.namespace)
    endif
```

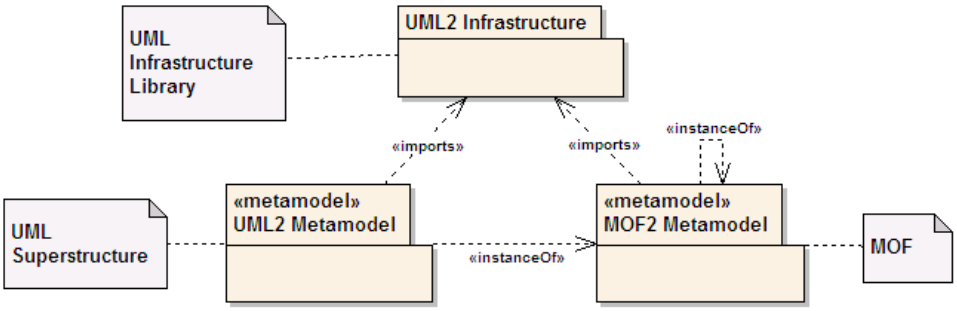

Figure 2.1: Dependencies between UML 2.0 and MOF 2.0 (extracted from [Nóbrega 06]).

An UML system model consists of three main views: (1) the functional view, which presents the system's functionality; (2) the static (or structural) view, presenting the system's structure (entities and relations between them); and (3) the dynamic view, which presents the internal behavior of the system.

UML is traditionally used as a metamodel, i.e., developers create models using the language established by UML. However, the UML Infrastructure Library also defines the UML Profile mechanism, which allows the user to introduce new notations or terminologies, providing a way to extend UML metaclasses to adapt them for different purposes. Profiles are a collection of Stereotypes, Tagged Values, and Constraints [OMG 06c, Silva 05a]. A Stereotype defines additional properties for UML elements, but these properties must not contradict the properties that are already associated with the UML element. Thus, a Profile can only extend UML by introducing new concepts into the language, but does not allow the user to edit the UML metamodel itself.

Although UML was definitely a step forward in setting a standard accepted and easily understood by the whole Software Engineering community and aligning it towards the

MDE paradigm, UML is still criticized for several reasons [Sellers 05, Thomas 04], such as: (1) being easy to use for software-specific domains (such as IT or telecom-style systems) but not for other substantially different domains, such as biology or finance; (2) not being oriented to how it would be used in practice; or (3) its development process being much more driven by politics than by software engineering needs. Nevertheless, it is important to note that UML is often the target of overzealous promotion which raises user expectations to an unattainable level, and the criticisms that follow afterward are usually influenced by this aspect [France 06].

2.1.2 XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) [XMI] is an OMG standard for exchanging, defining, interchanging, and manipulating metadata information by using the eXtensible Markup Language (XML) [XML a], and it can be used for any metadata whose metamodel can be specified in MOF. XMI is most commonly used as a format to exchange UML models between tools, although it can also be used for serializing models that are instances of other MOF-based metamodels. Therefore, XMI provides a way to map MOF to XML, which allows the mapping of any MOF-based metamodels – such as UML – to XML, providing an easy and very portable way to serialize and exchange models between tools.

Nevertheless, users often regard XMI as a last resort for exchanging models between tools. This is because it is very common for each tool to use its own vendor-specific "XMI extensions", which means that there is usually information lost during the process of exchanging models between different vendor tools.

2.1.3 Model-Driven Architecture (MDA)

The **Model-Driven Architecture (MDA)** [MDA] is a framework for the software development life cycle, and its main characteristic is the importance of models in the development process; the development process is driven by the activity of modeling the software system [Kleppe 03]. It is based on other OMG standards such as UML, MOF, Query/Views/Transformations (QVT) [OMG 05a] (which deals with model-to-model transformations), and XMI.

The core foundations of MDA consist of the following three ideas [Booch 04]: (1) *direct representation*, in which the focus of development should shift from the implementation technology to the problem itself; (2) *automation*, because it makes no sense to manually execute tasks that could easily be performed by automatic means; and (3) *open standards*, as they help eliminate gratuitous diversity.

Ideally, MDA generators should be able to build full applications from input models. One of the important concepts in MDA is the idea of multiple levels of models, and MDA defines two types of models: (1) the **Platform-Independent Model (PIM)** and (2) the **Platform-Specific Model (PSM)** [MDA, Kleppe 03]. A PIM is a model with a high level of abstraction that makes it independent of any implementation technology. This makes the PIM suitable to describe a software system that supports a certain business; the system is specified from the perspective of how it should support the business, without paying attention to implementation details (like specific relational databases or application servers). A PSM also specifies the system, but in terms of the implementation technology; it is easy to see that, given a PSM, only a developer who has knowledge about the platform corresponding to the PSM will be able to interpret that PSM correctly. A PIM can be transformed into one or more PSMs, each of those PSMs being designed for a specific technology; this is because it is very common for software systems today to make use of several technologies, so developers will be interested in obtaining a PSM per technology from a PIM (which is usually given to them by business experts).

Figure 2.2 presents an overview of MDA, with the concepts of PIM and PSM. The solid lines connecting the boxes are transformations, which are defined by transformation rules. MDA prescribes the existence of transformation rules, but it's up to the model designer to define what those rules are. In some cases the vendor provides rules as part of a standard set of models, profiles and transformation rules.

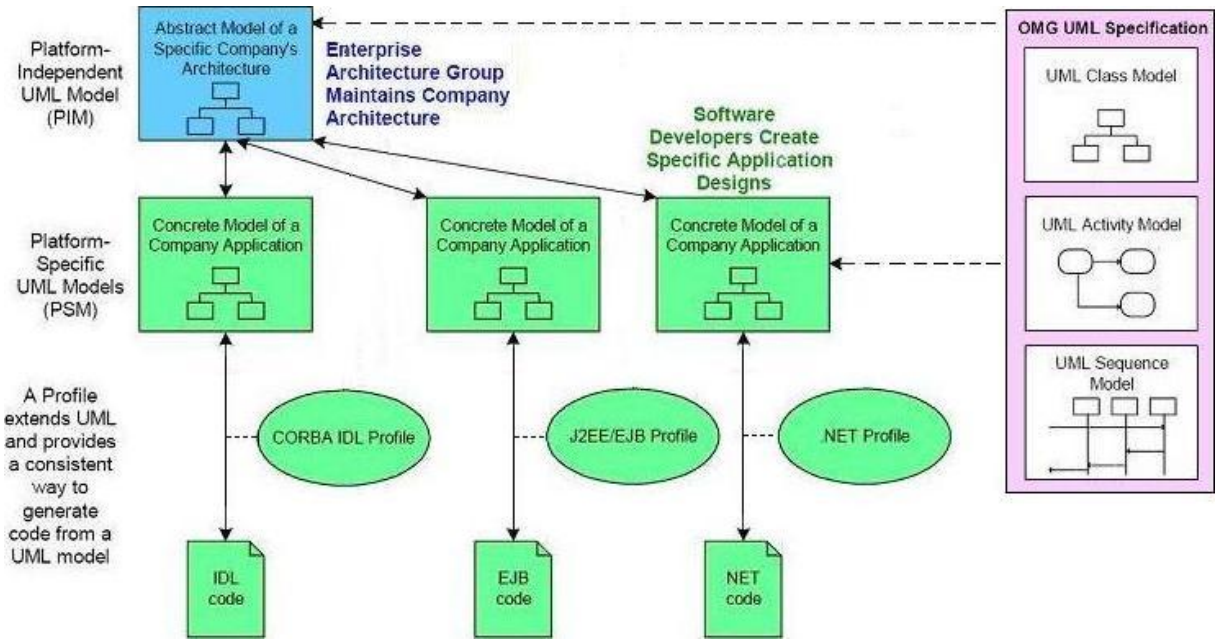

Figure 2.2: An overview of MDA (adapted from [Buchanan 02]).

2.2 UML Modeling Tools

This section presents some popular UML modeling tools: (1) ArgoUML [ArgoUML]; (2) Enterprise Architect [SparxSystems], from Sparx Systems; (3) Poseidon for UML [Gentleware], from Gentleware; and (4) Rational Rose [IBM], from IBM Rational. Although there are many other UML modeling tools currently available, this thesis presents this set of tools because their features and user-interfaces constitute a good representation of the current state of the art of this type of tools.

The analysis of these UML tools focuses on aspects such as: (1) adherence to standards; (2) ease of use; and (3) the set of available functionalities. Finally, a comparison between these tools is presented.

2.2.1 ArgoUML

ArgoUML 0.22 [ArgoUML] is an open-source UML modeling tool written totally in Java and released under the BSD license. Currently, this tool only supports UML 1.4, but its developers consider UML 2.0 as a goal to achieve. Although ArgoUML does not offer many of the functionalities that are typical of UML modeling tools, the tool is worth mentioning since it differentiates itself from the other tools of this kind due to its use of "cognitive psychology" [ArgoUML] to detect model inconsistencies and promote modeling best-practices. Figure 2.3 shows a screenshot of ArgoUML.

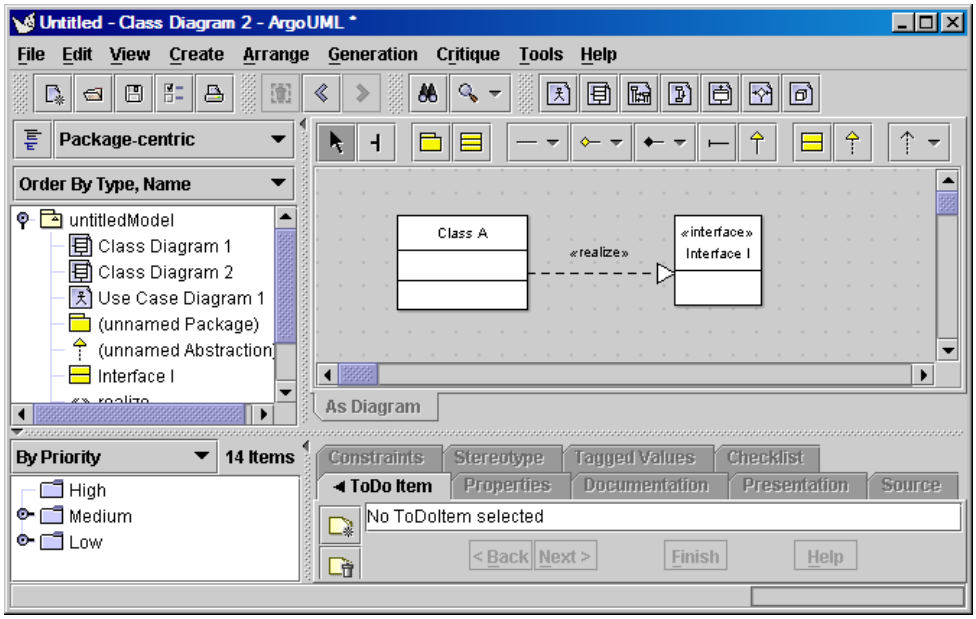

Figure 2.3: A screenshot of "ArgoUML".

ArgoUML supports the import/export to XMI, because its models are stored natively in that format. However, due to the fact that XMI started including information about the graphical representation of models only since UML 2.0, ArgoUML stores that graphical information separately from the model. Another interesting feature of ArgoUML is that it enforces user-defined OCL constraints, although this only happens for Classes and Features, because OCL versions before OCL 2.0 considered only these two concepts as allowable contexts to which constraints could be applied [ArgoUML].

This tool also offers code generation capabilities, but it does not provide any model transformations to support the various model abstraction levels of MDA.

2.2.2 Enterprise Architect

Enterprise Architect (EA) 6.1 [SparxSystems], from Sparx Systems, is a commercial tool that features good usability and a minimal learning curve. EA supports most of the UML 2.0, and is currently known through the Software Engineering community as one of the most versatile UML modeling tools available, offering a wide range of functionalities (such as Model Patterns) that considerably accelerate modeling tasks.

EA provides support for the UML Profile mechanism, and it makes the definition of an UML profile an intuitive task. However, the possibilities of UML profile definition are limited to specifying the generic syntax of the profile (e.g., defining stereotypes and what metaclasses they extend, or enumerations). Other semantic and syntactic relations, and constraints entered in the profile definition (using a text-based notation such as OCL), are not enforced when the user creates a model using that profile; the only validation that EA does enforce is the application of a stereotype to an instance of a metaclass (e.g., a stereotype that extends the metaclass *Association* cannot be applied to an instance of the metaclass *Class*). Figure 2.4 shows a screenshot of EA with an example of stereotype definition.

EA supports the import/export to XMI, and diagrams can be exchanged between different instances of EA, although the diagram exchange is done through an EA-specific mechanism (which means that diagrams cannot be totally exchanged between EA and other third-party tools, such as Rational Rose).

EA also offers code generation capabilities and some basic MDA-oriented model transformations, such as PIM-to-PSM. However, this tool does not allow users to define their own model transformations.

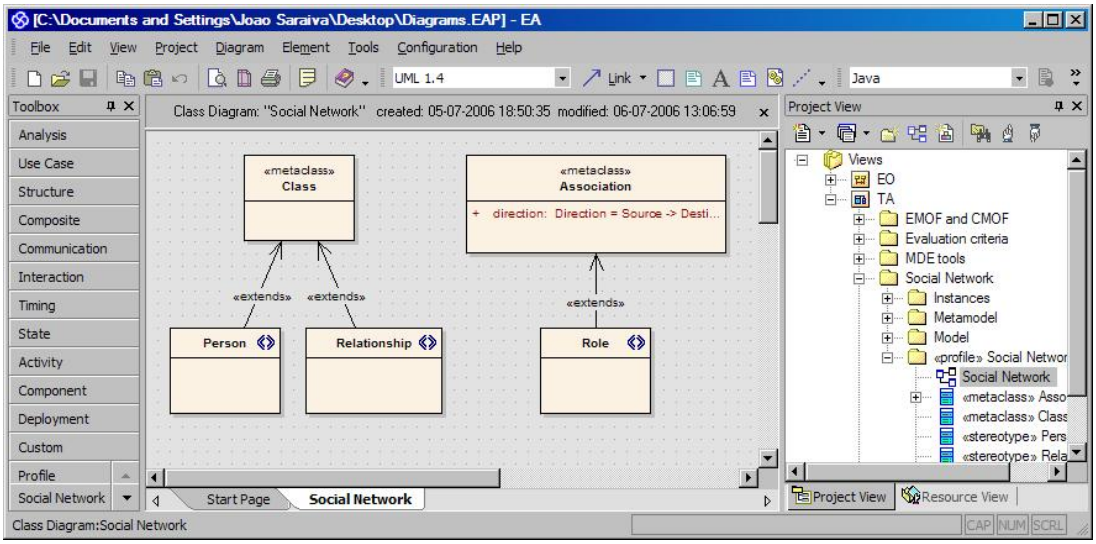

Figure 2.4: A screenshot of "Enterprise Architect".

2.2.3 Poseidon for UML

Poseidon for UML 4.0 [Gentleware], by Gentleware, is a commercial UML modeling tool that is based on ArgoUML. Unlike ArgoUML, this tool supports the UML 2.0, although this support is not yet complete. Additionally, this tool also features a wide range of functionalities, as is expected from a commercial tool. Figure 2.5 presents a screenshot of Poseidon for UML.

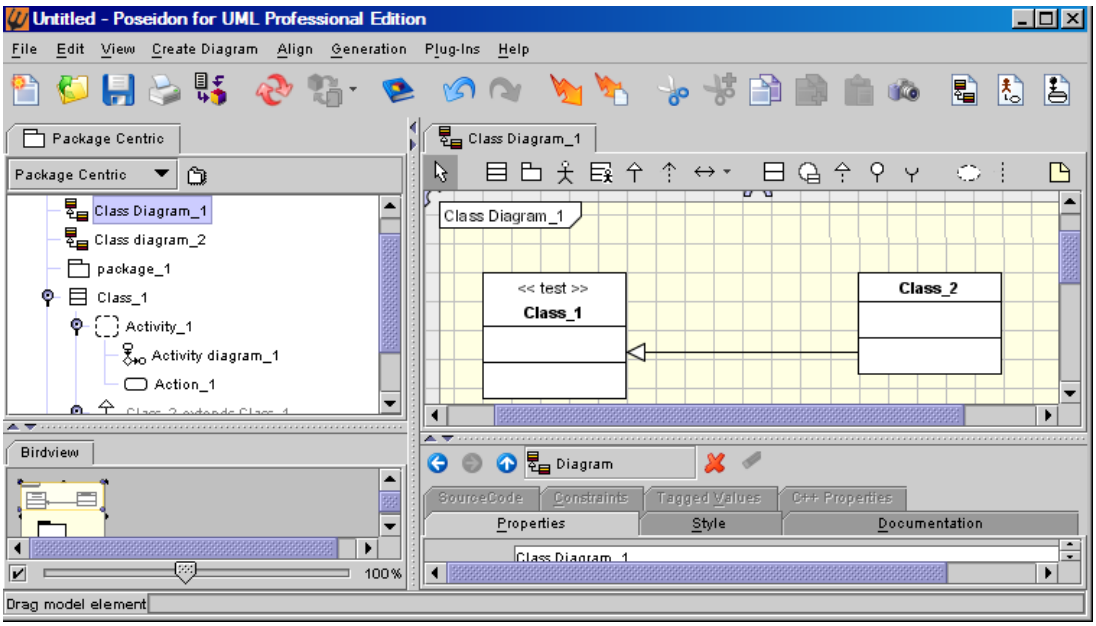

Figure 2.5: A screenshot of "Poseidon for UML".

Poseidon supports the import/export to XMI, because its models are stored in this format. However, the diagrams are also stored in XMI, because Poseidon supports UML 2.0 and the Diagram Interchange specification [OMG 06b]; this is unlike what happens with ArgoUML, which stores diagrams separate from the model. Additionally, Poseidon allows users to specify model constraints using OCL 2.0, but those constraints are not enforced by the tool (unlike ArgoUML).

Like ArgoUML, this tool also offers code generation capabilities, but it does not provide any model transformations to support the various model abstraction levels of MDA.

2.2.4 Rational Rose

Rational Rose 2003 [IBM], from IBM Rational, is a commercial UML modeling tool that is well known in the Software Engineering community. This tool supports the UML 1.x but not the UML 2.0, because the latter was only released in 2005 (two years after the release of Rational Rose 2003). Rational Rose also features a wide range of functionalities that promote modeling good-practices and accelerate the modeling tasks. Figure 2.6 shows a screenshot of Rational Rose.

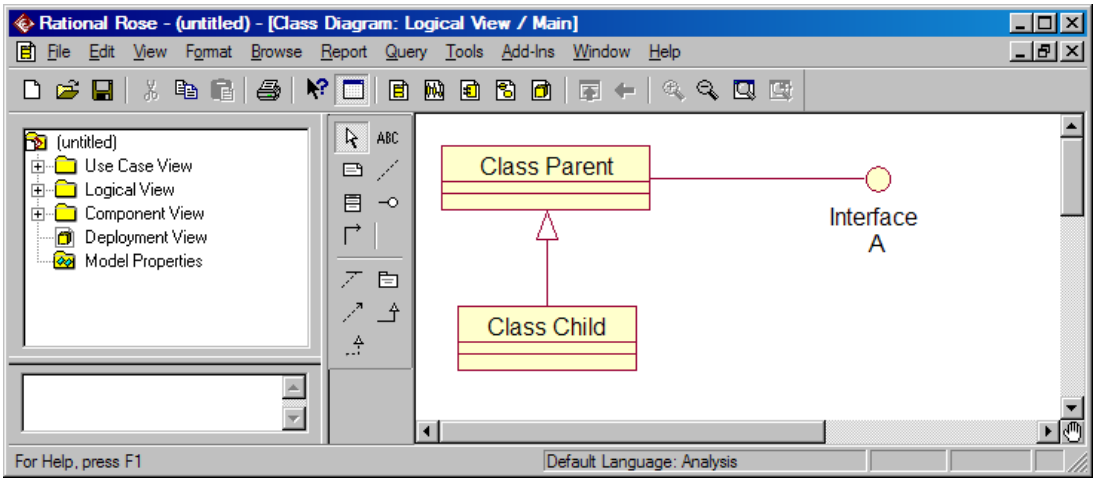

Figure 2.6: A screenshot of "Rational Rose".

Although Rational Rose includes the UML's standard stereotypes, it does not provide an easy way to add user-defined stereotypes, and there is no support for the definition of a profile. To add a stereotype, the user must use the configuration, scripting and menu-editing facilities of the tool, which is an error-prone process, as it also involves the manual editing of an INI file.

Rational Rose does not natively support the import/export of models to XMI, although there is a plugin by UniSys that adds this functionality to the tool. This same plugin

also allows the import/export of diagrams to XMI, although the generated XMI uses plugin-specific tags to represent diagrams.

This tool also offers code generation capabilities, but it does not provide any model transformations to support the various model abstraction levels of MDA.

It should be noted that Rational Rose has recently been replaced by Rational Software Modeler [Mittal 05]. This replacement was made in the scope of IBM's Rational Software Development Platform, which is a common development environment, based on Eclipse [Eclipse]. This platform is shared by other IBM products (such as the Rational Web Developer, the Rational Application Developer, the Rational Software Architect, the Rational Functional Tester, and the Rational Performance Tester), and allows the integration of all these products.

2.2.5 Comparison Between the Analyzed Tools

After analyzing these tools, it was important to compile a list containing the most interesting and important functionalities that the tools provided.

Note that the objective of this comparison is to determine a set of features that ProjectIT-Studio/UMLModeler users would likely find interesting and useful in the context of ProjectIT-Studio. Nevertheless, some aspects (e.g., generation of source-code or documentation) that would normally be important in stand-alone tools like the ones presented in this section are not mentioned because ProjectIT-Studio has already defined how they will be handled (as is explained in the next chapter, "ProjectIT-Studio Context").

This list consists of the following features:

1. *Copy Diagram To Clipboard*: allows the user to place screenshots of diagrams in the clipboard, for any kind of purposes;

2. *Save Diagram As File* (related to the previous feature): allows the user to save screenshots of diagrams in an image file;

3. *Create Pattern*: allows the capture of certain recurring patterns (like the ones from [Gamma 95]) that the user keeps applying;

4. *Create Classes From Patterns* (related to the previous feature): allows the application of patterns to a model;

5. *Provides "Model Overview" Tree*: provides a way to quickly get an overview of the model, by presenting it in a tree structure;

6. *Create Diagram Elements From Model Overview Tree* (related to the previous feature): allows the user to place an already-existing element into a diagram, by dragging it from the Model Overview tree to the diagram;

7. *Create Profile*: allows the definition of a profile, as a collection of user-defined elements that extend the UML language (i.e., stereotypes);

8. *Create Stereotype* (sometimes related to the previous feature): allows the definition of a stereotype, as an element that extends the UML language (note that a tool that supports the creation of stereotypes does not necessarily support the creation of profiles);

9. *Custom Icons For Stereotypes* (related to the previous feature): allow the definition of alternative representations for a stereotype;

10. *Supports UML 2.0*: this feature is important because UML 2.0 is the standard at the moment;

11. *Supports User-Defined Model-to-Model Transformations*: allows users to define their own model-to-model transformations, and later apply them to a model;

12. *UML Standard Stereotypes*: allow users to apply the stereotypes defined in the UML Superstructure specification;

13. *XMI Import/Export*: allows the achievement of interoperability between different tools;

14. *Enforces User-Defined Constraints*: allows users to provide their own constraints (into the model, a profile, or a stereotype) and the tool will ensure that the model is consistent with all of those constraints.

Table 2.1 presents an overview of the main features offered by the UML modeling tools analyzed in this section, as well as the support that each tool offers to these features.

Some interesting conclusions can be taken from this analysis: (1) all tools allow the export of a diagram to the clipboard or a file; (2) all tools provide a tree-based "model explorer"; (3) all tools allow the creation of stereotypes, although only some tools allow alternative icons; (4) it is relatively rare to find a tool that enforces user-defined model constraints; (5) none of the tools allow the specification of user-defined model-to-model transformations; (6) all tools provide a set of the UML's standard steretypes; and (7) all tools support the import/export to XMI.

Features \ Tools	ArgoUML	Enterprise Architect	Poseidon for UML	Rational Rose
Copy Diagram To Clipboard	No	Yes	Yes	Yes
Save Diagram As File	Yes	Yes	Yes	Yes (through Web Publishing)
Create Pattern	No	Yes	No	No
Create Classes From Patterns	No	Yes	No	Yes
Provides "Model Overview" Tree	Yes	Yes	Yes	Yes
Create Diagram Elements From Model Overview Tree	Yes	Yes	Yes	Yes
Create Profile	No	Yes	Yes (by creating a Plugin)	No
Create Stereotype	Yes	Yes	Yes	Yes
Custom Icons For Stereotypes	No	Yes	No	Yes (through .ini file)
Enforces User-Defined Constraints	Yes, using OCL (only for Classes and Features)	No	No	No
Supports UML 2.0	No	Yes	Yes	No
Supports User-Defined Model-to-Model Transformations	No	No	No	No
UML Standard Stereotypes	Yes	Yes	Yes	Yes
XMI Import/Export	Yes	Yes	Yes	Yes (through UniSys plugin)

Table 2.1: Comparison between the UML tools analyzed.

2.3 Technological Frameworks

This section presents the technological frameworks that are relevant to this work: (1) the Microsoft .NET Framework [.NET a]; (2) the IKVM.NET framework [IKVM.NET]; (3) the Eclipse.NET platform [Eclipse.NET]; (4) the Eclipse Graphical Editing Framework (GEF) [EclipseGEF b]; (5) the nUML project [nUML]; and (6) the Eclipse UML2 project [EclipseUML2].

Although there are other implementations of the UML metamodel, the Eclipse UML2 and nUML projects are the only ones that fulfill the following requirements: (1) support UML 2.0; and (2) their licenses present no objections to being used in ProjectIT-Studio (both projects are open-source and do not have such integration restrictions). Because of this, both projects are thoroughly analyzed in this thesis.

2.3.1 Microsoft .NET Framework (.NET)

The **.NET Framework** [.NET a, .NET c] is a platform that supports the development and execution of command-line or form-based applications, web applications, and web services. This platform provides an execution mechanism, called Common Language Runtime (CLR), and a class library to support software development, designated ".NET Framework Class Library". The .NET Framework architecture and the relationship be-

tween the CLR and the surrounding operating system and applications are presented in Figure 2.7.

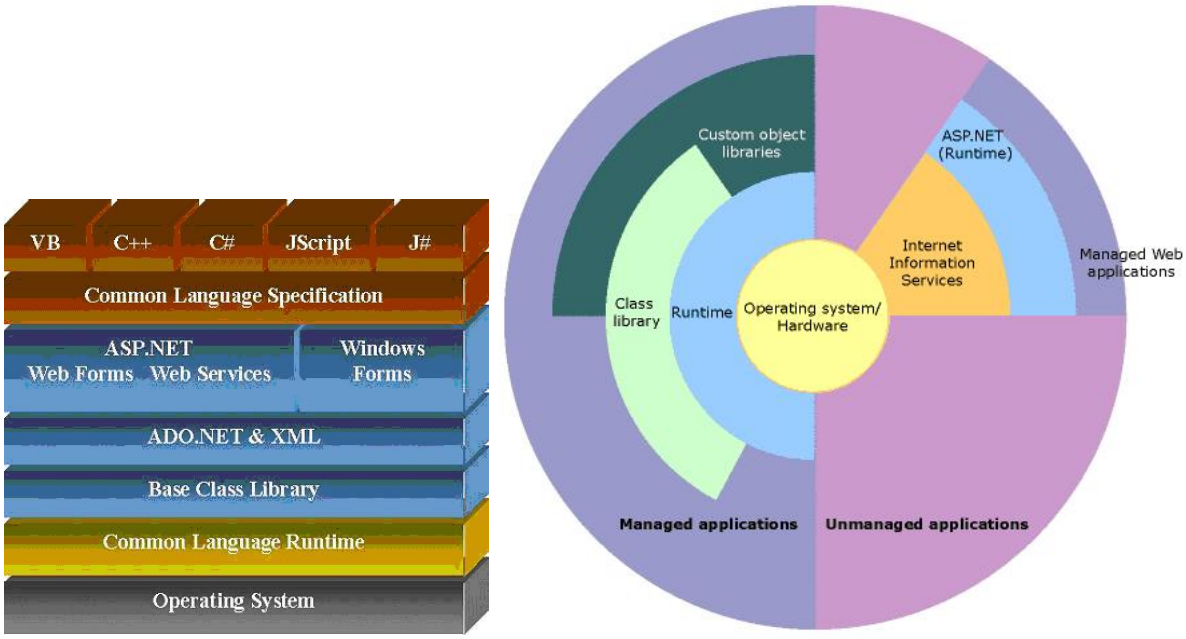

Figure 2.7: The .NET Framework architecture (extracted from [.NET b] and [.NET c]).

The CLR manages the application's code execution, converting instructions in Microsoft Intermediate Language (MSIL) (the language to which source code is initially compiled) to the corresponding native machine-code instructions in real-time. The CLR is also responsible for core services that provide memory management for processes and threads, as well as offering exception handling, runtime compilation, and reflection mechanisms that can be enhanced by using remote method invocation functionality, while still assuring a high security and robustness level.

Source code management is crucial to the .NET platform, and that's the main reason for the existence of managed and unmanaged code concepts. The difference between the two concepts lies on whether code is aimed at the CLR provided environment's functionality or not, respectively.

The .NET Framework Class Library is another important component of this platform, as it provides a collection of classes that can be used to develop software applications. These out-of-the-box classes provide typical I/O functionality, string manipulation, communication mechanisms, user interface creation, collection management, and other typical programming languages abstractions to ease the developer's tasks.

With the .NET framework, it is possible to build a wide range of software applications, ranging from simple command line applications or "fat clients" using Graphical User

Interface (GUI) controls, to complex web applications based on ASP.NET technology [ASP.NET] (using WebForms and XML-based interoperable Web Services).

The .NET Framework is currently in version 2.0, which presents several improvements when compared with the previous 1.1 version [.NET d]. Along with this new version of the .NET Framework came the C# 2.0 language, which presents some new features, such as: (1) generics, which allow the type-safe configuration of collection classes, enabling developers to achieve a higher level of code reuse and enhanced performance when dealing with typical collection classes; (2) partial classes, which are used to split large classes in several files (this can be very useful to separate automatically-generated code from user-defined code, for example); (3) nullable types, which allow a value-type variable to contain a null value; (4) anonymous methods, which supports the lambda calculus (i.e., to pass a block of code without a name as a parameter anywhere a method is expected, instead of defining a new named method); (5) static classes, which provide a safe and convenient way of declaring a class containing static methods that cannot be instantiated; and (6) property accessor accessibility, which allows the definition of different levels of accessibility for the get and set accessors on properties.

In conclusion, the major benefit of the .NET Framework derives from the improved scalability and performance of the applications that can be developed with this platform, and from the restructuring of the ASP.NET technology to the new ASP.NET 2.0, which features new types of web controls and a better support for various browsers and computational systems.

2.3.2 IKVM.NET

The **IKVM.NET tool** [IKVM.NET] is a Java Virtual Machine (JVM) [Java] for the .NET platform. It appeared in a peculiar context and time when most of the software community believed that Java and .NET technologies were mutually exclusive. The goal of IKVM.NET is to reduce the gap between these two technological platforms by providing an interoperability mechanism to overcome this pitfall. It mainly offers a variety of integration patterns between the Java and .NET languages and platforms, by applying virtual-machine-related technologies for byte-code translation and verification, and class loading. Figure 2.8 illustrates how this tool works and how it achieves interoperability between the two platforms.

This tool has two operation modes: (1) static mode, in which Java byte-code is previously translated to MSIL and compiled into a .NET assembly that can be used by .NET applications (thus providing a mechanism for using objects defined in the Java platform as if they were .NET objects); and (2) dynamic mode, in which Java classes and JAR files are

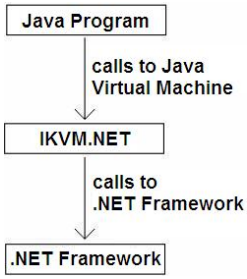

Figure 2.8: How IKVM.NET achieves interoperability between Java and .NET.

directly used, without any previous conversion, to execute Java applications on the .NET runtime environment (Java byte-code is translated to MSIL instructions in runtime).

The IKVM.NET tool is composed by three applications: (1) **ikvm**, which supports the dynamic mode; (2) **ikvmc**, which supports the static mode; and (3) **ikvmstub**, which is required to create .NET stubs from .NET assemblies, thus allowing Java code to be compiled by referencing .NET code.

The *ikvmstub* application is a tool to generate JAR files that act as stubs (or "empty representations") of .NET assemblies, so that Java code can be compiled by referencing .NET code. This feature is very useful since it allows Java code to use .NET functionality, by just referencing the "empty" classes available in the stub JAR file. The other IKVM.NET applications then detect API calls to the stubs and replace them with invocations to the "real" .NET functionality.

The *ikvm* application is a starter executable used in dynamic mode to play the role of the JVM (i.e., it allows the execution of a Java program on the .NET framework without any previous transformation of the Java classes). This functionality is achieved by on-the-fly conversion of Java's API calls to .NET API calls at runtime. This tool also detects API calls to .NET assembly stubs, and replaces those calls with the corresponding calls to the .NET functionalities represented by those stubs.

The *ikvmc* application is the "static mode" compiler, used to compile Java classes and JAR files into a .NET assembly (EXE or DLL, depending on whether any of the Java classes provides a "main" method). Like the *ikvm* application, this tool also detects API calls to .NET assembly stubs, and replaces those calls with the corresponding calls to the .NET functionalities represented by those stubs.

2.3.3 Eclipse.NET

Eclipse.NET [Eclipse.NET] is an open-source project that aims to deliver an extensible, open platform to support the development of integrated tools for the .NET runtime environment. It is a plugin-based application and framework that eases the process of

building, integrating and using software tools within the .NET execution environment. The default configuration of the Eclipse.NET platform includes a set of plug-ins that extends the platform with some features typically found in an IDE, but it is more than an IDE. It is inspired by the success of the Eclipse project [Eclipse] in delivering a similar platform for the Java execution environment.

Eclipse.NET features a plugin-based architecture that focuses on modularity and plugin collaboration, while allowing full control over plugin dependency relationships (i.e., developers have full control to specify dependencies between the developed plugins). Its main architectural concepts are: (1) the core **Platform Runtime**, (2) the **plugin** and (3) the **extension point** [Saraiva 05a].

The *Platform Runtime* is responsible for configuring the Platform, detecting available plugins, reading their manifest files, building a plugin registry, and performing some validation tasks, such as detecting extensions to extension points that do not exist. After all these tasks have been completed, the Runtime makes the registry and the Platforms configuration available to plugins.

An *Eclipse.NET Plugin* is Eclipse.NET's smallest unit of behavior. Any tool can be developed as a plugin or as a set of plugins. In fact, all of the Platform's functionality is located in plugins, the exception to this being the functionality in the core Platform Runtime. A plugin also features a manifest file which contains information about the plugin itself, such as extension points (explained below) that the plugin provides, and what extensions the plugin provides to extension points defined by other plugins. This manifest file is located separately from the plugin itself, because Eclipse.NET's plugin-loading mechanism uses the **Lazy Load design pattern** [Fowler 03], so the Runtime can read the information about the plugin without actually loading the plugin into memory. This way, the Runtime can build the registry without loading every plugin, thus saving resources. Plugins can also depend on each other. These dependencies are expressed in the plugin's manifest file. This way, a plugin will only be used (and, of course, successfully loaded) if all other plugins on which it depends are also successfully loaded.

An *Eclipse.NET Extension Point* is a way for plugins to inter-operate with each other. This interconnection model is achieved through the classical process: (1) any plugin can declare any number of extension points; (2) on the other hand, any plugin can provide any number of extensions for one or more extension points declared by other plugins. Figure 2.9 illustrates how plugins and extension points are related. Plugin 1 declares three extension points (Extension Point 1, Extension Point 2, and Extension Point 3). Extension Point 1 and Extension Point 2 receive the extension (or contribution) from Plugin 2. On the other hand, Plugin 2 also declares two other extension points (Extension Point 4 and Extension Point 5). These last two extension points may also receive contributions from

other existing plugins. Plugin 1 may also be contributing to other existing extension points, not represented in the figure.

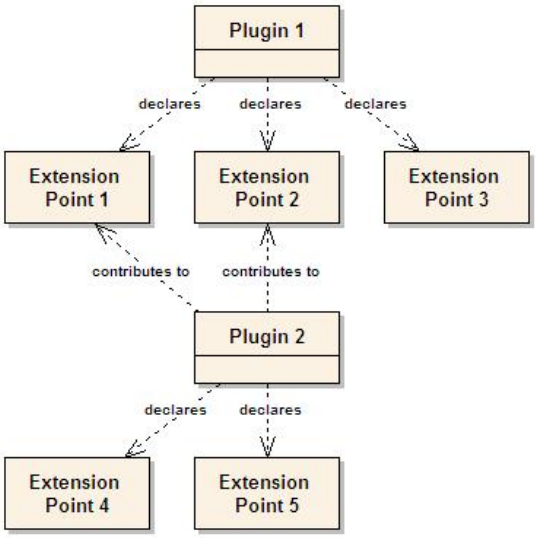

Figure 2.9: Plugins can declare, and contribute to, extension points (extracted from [Saraiva 05a]).

Eclipse.NET was created in a previous work [Saraiva 05b] with the main purpose of being an integration platform on which ProjectIT-Studio (as well as any other platforms) could be based [Saraiva 05a].

As in the Eclipse project, Eclipse.NET's graphical environment is still based on the **Standard Widget Toolkit (SWT)** [EclipseSWT]. SWT is a platform-specific run-time that offers **widget** objects (such as text boxes, scroll-bars, and other familiar user-interface controls) to Java developers, but underneath uses native code calls to create and interact with those controls, making SWT-based applications essentially indistinguishable from native platform applications.

The tree of SWT widgets is primarily based on the **Composite design pattern** [Gamma 95, Gamma 03], whose structure is illustrated in Figure 2.10; this pattern is applied to SWT by defining a special type of widget, called `Composite`, which can contain other widgets; this means that SWT allows a developer to create user-interfaces with a varying degree of complexity by simply creating a tree of SWT widget instances. Examples of composite widgets include the traditional `Tree` or `List` widgets. Figure 2.11 illustrates how this design pattern is applied to SWT.

2.3.4 Eclipse Graphical Editing Framework (GEF)

The **Eclipse Graphical Editing Framework (GEF))** [EclipseGEF b, EclipseGEF a] is an open-source framework dedicated to providing a rich and consistent graphical editing

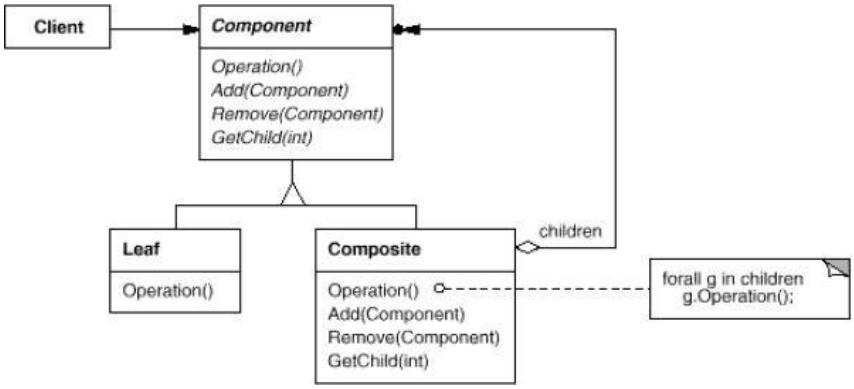

Figure 2.10: The Composite design pattern's structure (extracted from [Gamma 95]).

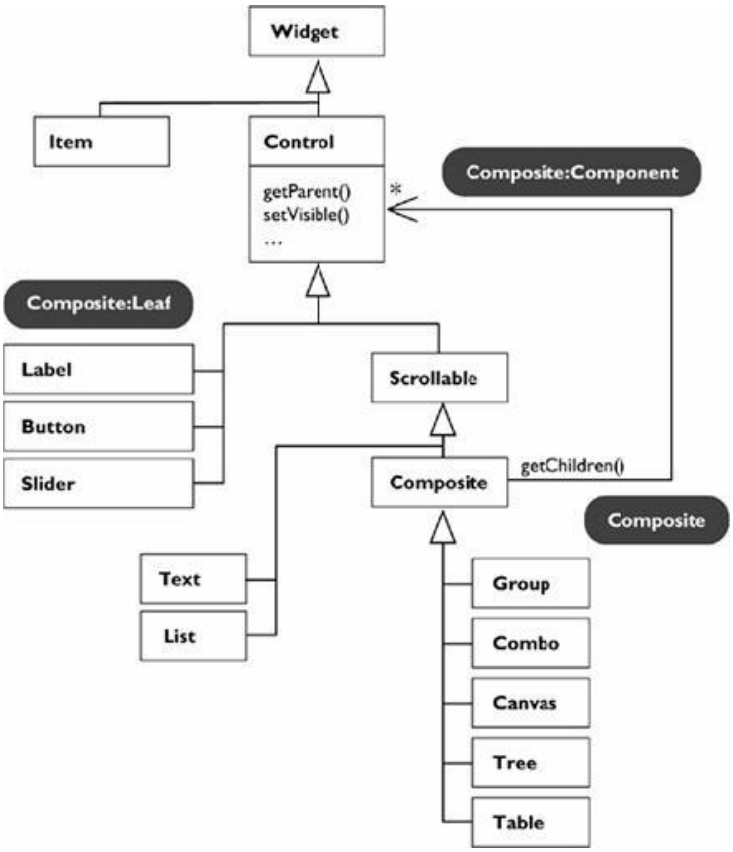

Figure 2.11: The Composite design pattern applied to SWT (extracted from [Gamma 03]).

environment for plugins on the Eclipse Platform [EclipseGEF b]. GEF allows the developer to create graphical editor plugins based on the **Model-View-Controller pattern** [Fowler 03].

The MVC pattern is often used by applications that need to maintain multiple views of the same data (or *model*). This pattern categorizes objects as one of three types: (1) **Model** for maintaining data; (2) **View** for displaying the data; and (3) **Controller** for handling events that affect either model or views. Because of this separation, the

same model can be used with multiple views and controllers, and new types of views and controllers can also use that model without forcing a change in the model design.

GEF consists of two plugins, illustrated in Figure 2.12, for the Eclipse Platform: (1) the **Draw2D** plugin; and (2) the **GEF** plugin (after which the framework itself is called).

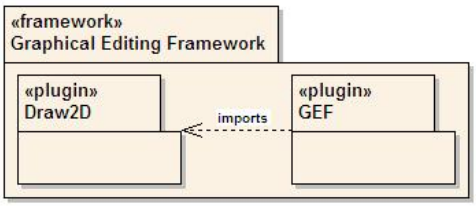

Figure 2.12: The plugins that compose the Graphical Editing Framework (GEF).

The Draw2D plugin essentially consists of a lightweight widget system hosted on a SWT Canvas, which is a widget that provides a generic drawing surface that can be used like a traditional drawing board. This plugin provides developers with the ability to graphically display anything they want by drawing **Figures**, which are the graphical building blocks of Draw2D. Developers can use some standard shape figures, such as `Ellipse`, `Polyline`, `RectangleFigure` or `Triangle`, or create their own figures. Additionally, some figures can act as an overall container for child figures, allowing the construction of relatively complex figures (by using the Composite design pattern). Figure 2.13 presents a tree of figures and the respective representation if each figure is painted as a full rectangle.

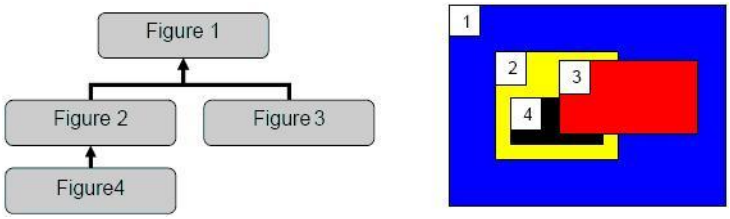

Figure 2.13: A tree of Figures and their representation (extracted from [EclipseGEF a]).

The GEF plugin, on the other hand, is responsible for implementing the MVC framework itself. Although GEF does not provide any mechanism for defining the Model component (thus allowing the developer to use any possible model), GEF does provide mechanisms for defining both the View and the Controller components, as Figure 2.14 illustrates.

The View component is defined by using Draw2D Figures; there is one figure per model object, although that figure can contain other figures (e.g., figures representing fields of the model object). Although it is possible to define a figure by using the corresponding model object (i.e., by allowing the figure constructor to receive the corresponding model

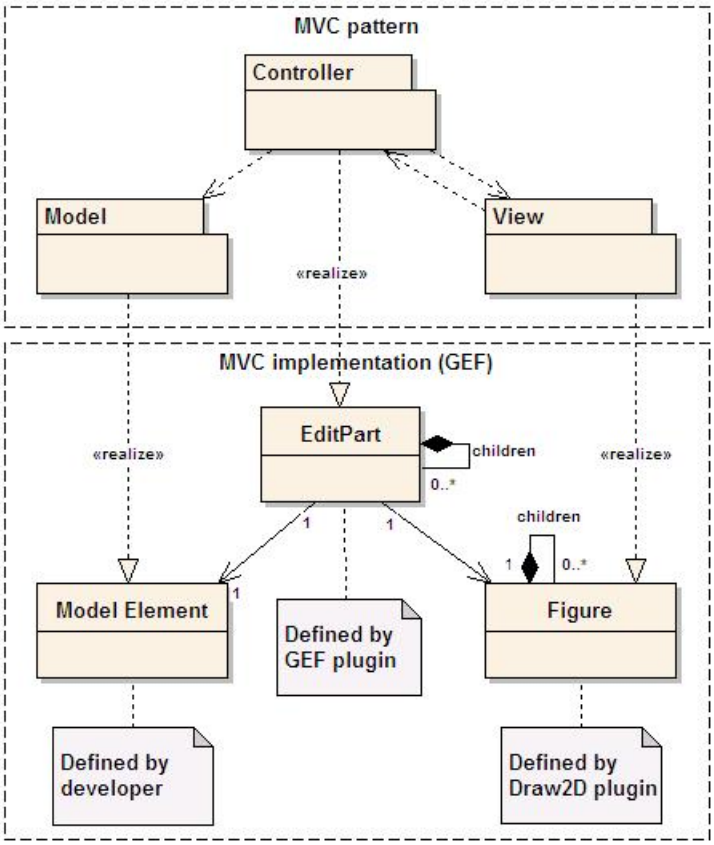

Figure 2.14: GEF is a framework oriented towards the MVC pattern.

object directly), it is generally considered good practice to completely separate the View and Model implementations and leave the mapping between them to the Controller, as this mapping may be non-trivial; otherwise, if this was done in the View, it would likely originate situations in which logic was embedded in the View definition, which is usually not desired.

The Controller component is defined by using **EditParts**. An EditPart is responsible for: (1) linking a model object to its visual representation; (2) making changes to the model, when the proper user input is received; and (3) detecting changes to the model and updating the view accordingly. Most interaction with EditParts is done by using **Requests**. Requests specify the type of interaction, and are used in various tasks, such as targeting, filtering the selection, graphical feedback, and most importantly, obtaining **Commands** (which are the implementation of the **Command design pattern** [Gamma 95]). A Command is used to encapsulate an user action as an object, providing the ability to support undoable operations. Figure 2.15 illustrates how Commands are obtained and how they are used: (1) an EditPart is asked for the Command corresponding to the given Request; (2) the Command is returned and executed; and (3) the execution of the Command changes the model. Although not represented in Figure 2.15,

the changes made to the model will be detected by the corresponding EditPart, which will then update the view accordingly.

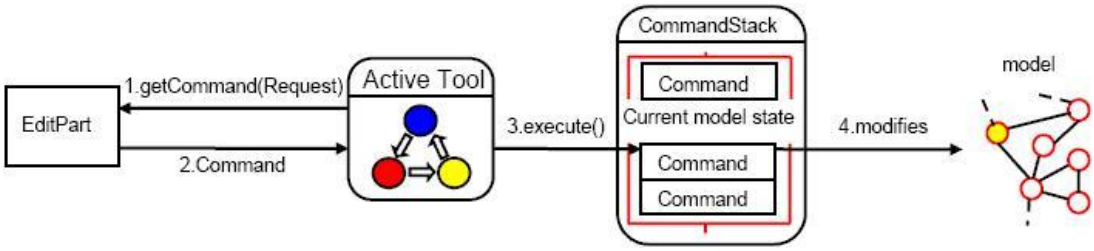

Figure 2.15: Commands are created by EditParts to modify the model (extracted from [EclipseGEF a]).

2.3.5 nUML

nUML [nUML] is a .NET library for manipulating UML 2.0 models, supporting serialization to and from XMI 2.1. Like the Eclipse UML2 project, it's goal is to provide an infrastructure for creating and manipulating UML 2.0 models.

This project is still in its early stages and thus still suffers from a lack of crucial features, such as: (1) the inability to apply a stereotype to an Element (because this mechanism is not clearly described in the UML specification); (2) the UML elements do not support the **Observer design pattern** [Gamma 95], which would be required to alert any interested parties that the state of an Element has changed; and (3) there is no type-checking on any of the operations available in each Element.

These features are essential to a well-structured modeler, with special relevance to the usage of the Observer pattern. This pattern is very important in the context of model editors (such as the ProjectIT-Studio/UMLModeler tool) because it allows the decoupling of editor logic from model logic (i.e., the editor does not have to know internal details about the behavior of the model). This simplifies the creation of the editor, since it only has to add itself as an "observer" of the model and interpret events as they appear.

Although the project itself seems promising, the current lack of these features was considered very serious, and it was the reason why it was decided not to use the nUML library in this work.

2.3.6 Eclipse UML2 project

The **Eclipse UML2** project aims to provide an UML 2.0 implementation library based on the Eclipse Modeling Framework (EMF) [EclipseEMF], which is a modeling frame-

work and code generation facility for building tools (and other applications) based on a structured data model specification.

The purpose of this UML 2.0 implementation is to provide an infrastructure that allows Eclipse plugins (or even other Java applications) to create and manipulate UML models. The Eclipse UML2 project is still in development, with its implementation covering the majority of the UML 2.0 metamodel.

Applications that are not based on Eclipse can also use this library, although they must also use some other Java-based libraries (namely the libraries required by EMF) due to some project dependencies. The size of the UML2 and the EMF libraries themselves is in the order of the megabytes (MB). However, the Java platform does not load library (JAR) files as a whole, but rather only accesses the class files contained in those libraries, and such loading operations occur only when necessary. This means that, in practice, the loading of the UML2 library and its dependencies occurs gradually over time, thus not affecting application performance in a way that is noticeable by the application user.

This library can also be used in the .NET environment, by using the IKVM.NET *ikvmc* tool for static compilation of such libraries from Java to .NET. Nevertheless, this particular case comes at a relatively heavy cost, as the resulting .NET library is a single file that has a size also in the order of the megabytes; additionally, the .NET Framework loads libraries and their dependencies as a whole (unlike the Java platform, which loads class files when necessary instead of loading entire libraries directly). This means that, when this library is loaded by a .NET application, there will be a significant performance penalty (because the application's process must allocate a significant amount of memory and the entire library must be loaded into memory) which will be noticed by the application user.

Despite this performance problem in the .NET environment, the Eclipse UML2 library was initially used in this work. However, because of some problems that surfaced during the implementation of ProjectIT-Studio/UMLModeler, it was decided to stop using this library. The rationale for all the decisions regarding Eclipse UML2 in the context of this work are explained in Chapter 4.

Chapter 3

ProjectIT-Studio Context

This chapter presents **ProjectIT-Studio**, which provides the context for the development of the UML modeling plugin. ProjectIT-Studio is presented from a number of different viewpoints: (1) the usage scenarios that it is meant to support; (2) its component architecture, based on Eclipse.NET plugins; and (3) the kind of support that ProjectIT-Studio offers to MDA.

3.1 ProjectIT-Studio Usage Scenarios

ProjectIT-Studio is meant to support two primary usage scenarios: (1) *stand-alone* (single-user), and (2) *collaborative work* (multi-user). Figure 3.1 illustrates these scenarios.

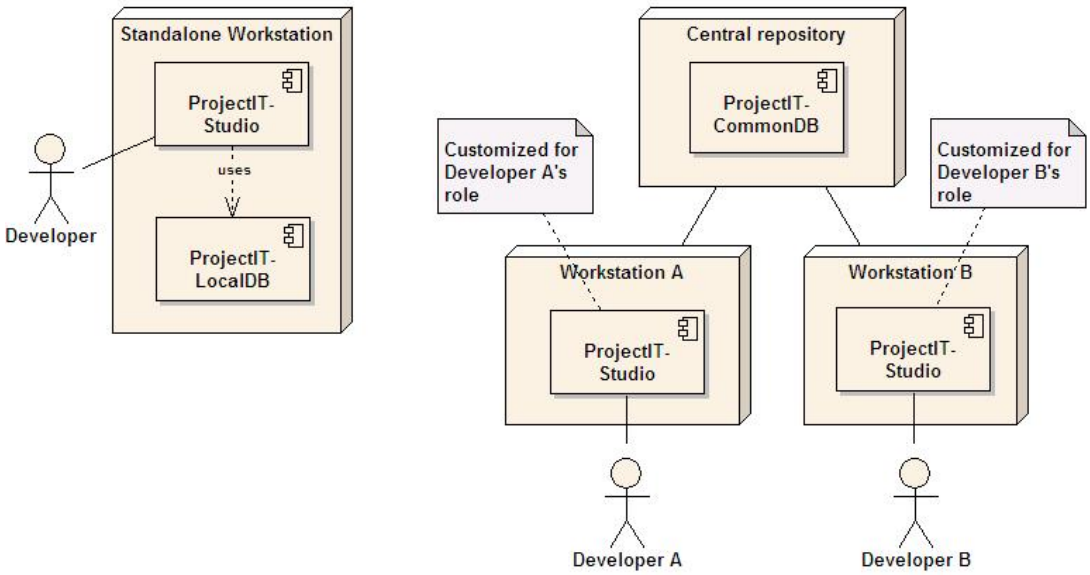

Figure 3.1: ProjectIT-Studio usage scenarios (extracted from [Silva 06a]).

The *stand-alone* scenario is typically associated with development in an environment disconnected from the Central Repository that contains the ProjectIT-CommonDB component, with which all instances of ProjectIT-Studio communicate; in this scenario, all information is stored locally by ProjectIT-Studio, and can be synchronized with the Central Repository at a later time. On the other hand, the *collaborative work* scenario is associated with an environment in which the Central Repository is available (e.g., if the Central Repository can be accessed through the Internet and the workstation with ProjectIT-Studio is also connected to the Internet).

It should be stressed that, independently of these scenarios, ProjectIT-Studio is used differently depending on the interacting role (i.e., Architect, Requirements Engineer, Designer or Programmer), according to the ProjectIT approach.

3.2 ProjectIT-Studio Architecture

ProjectIT-Studio consists of an application based on Eclipse.NET, which runs on top of the .NET Framework. Furthermore, ProjectIT-Studio can be seen as an orchestration of three different plugins (ProjectIT-Studio/Requirements, ProjectIT-Studio/UMLModeler and ProjectIT-Studio/MDDGenerator [1]), as Figure 3.2 illustrates.

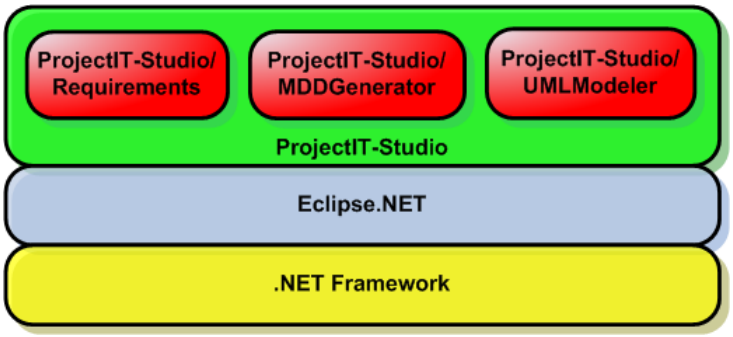

Figure 3.2: ProjectIT-Studio main components (extracted from [Silva 06a]).

One of the requirements during the development of ProjectIT-Studio was that its constituent plugins should be relatively independent among themselves (i.e., each plugin should not require the presence of other plugins to operate properly). This requirement adjusts well to Eclipse.NET's architecture, as its plugins can *require* other plugins (through the "import" relationship) or *work with* – without requiring – other plugins (through extension points). The ProjectIT-Studio plugins (Requirements, UMLModeler,

[1] For text simplicity, the "ProjectIT-Studio" prefix is usually omitted in the remainder of this thesis. For example, any mentions to "Requirements" will mean a reference to ProjectIT-Studio/Requirements (unless explicitly stated otherwise).

and MDDGenerator) were implemented as a set of Eclipse.NET plugins, which are going to be described next.

3.2.1 ProjectIT-Studio/Requirements

ProjectIT-Studio/Requirements is the plugin responsible for requirements engineering issues. The supporting vision of this plugin is that it should be used for "writing requirements documents, like a word processor, and as requirements are written, it will detect errors that violate the requirements language rules and provide the appropriate warnings" [Ferreira 06].

The implemented plugin provides a specific requirements text editor that supports all the typical IDE features such as on-the-fly syntactic verification and syntax highlighting, which means that all expressions are validated as soon as they are written, and errors are immediately detected and highlighted. The auto-complete feature is also always available presenting hints of how to complete the sentence.

The plugin follows the emerging Domain-Specific Language (DSL) [MartinFowler, DSMForum] approach by supporting the ability to define new languages to better tackle the problem at hands, thus raising the abstraction level at which developers work.

3.2.2 ProjectIT-Studio/UMLModeler

ProjectIT-Studio/UMLModeler is a plugin for standard UML modeling in the context of the ProjectIT approach. It allows the creation of visual models of the system using the UML 2.0 modeling language [OMG 05b]. The Designer creates the models based on a given UML profile (e.g., the XIS2 UML profile [Silva 07]). These models are then used by MDDGenerator to create the system artifacts according to specific software architectures.

In addition to the creation of UML models, the plugin also features a simple Profile definition mechanism, which allows further customization of UML; this allows the Designer to further adapt the system modeling language (i.e., the UML profile) to the categories of templates that MDDGenerator will use.

A profile definition consists of two separate components: (1) the *syntax* and (2) the *semantics* of the profile. The syntax is defined by associating stereotypes with images, which is the typical mechanism that the UML specification provides [OMG 05b] and the most common among UML modeling tools; this association between stereotypes and images is done entirely within the plugin. As for the definition of profile semantics, the plugin supports the graphical specification of extensions between Stereotypes and UML metaclasses (resembling Enterprise Architect [SparxSystems]). The plugin also supports the definition of additional profile semantics, using external .NET assemblies containing

code to validate the model in terms of the profile; this allows it to assure the continuous validation of the model being created, according to both the UML semantics and the profile's semantics.

The UMLModeler plugin was developed in the context of this work and is described in greater detail in the next chapter.

3.2.3 ProjectIT-Studio/MDDGenerator

ProjectIT-Studio/MDDGenerator is the plugin responsible for the generation of system artifacts based on models. It relies on the concepts illustrated in Figure 3.3: (1) Model; (2) Software Architecture; and (3) Template.

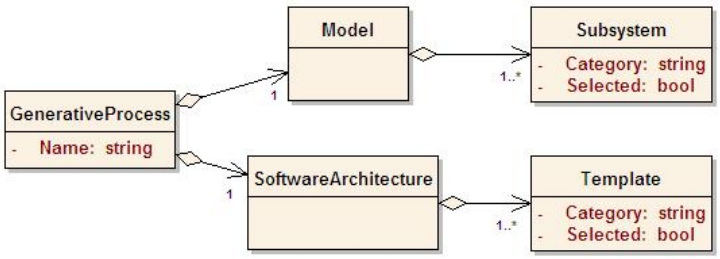

Figure 3.3: Overview of the MDDGenerator concepts (extracted from [Silva 06a]).

According to the ProjectIT approach, a **Model** is an abstract representation of a software system, created by the Designer and based on a given UML profile (e.g., the XIS2 UML profile); a model can be further divided into subsystems, which are logical divisions of the model that pertain to a category such as entities, actors or interaction spaces. A **Software Architecture** is a generic representation of a software platform, created by the Architect when developing templates for that target platform. **Templates** are generic representations of software artifacts to support the ProjectIT approach's "Model2Code" transformations, and they pertain to a specific category such as database, data services, user interfaces, and others.

The input of this plugin is a generative process, which establishes a mapping between the system model and the software architecture of the final application. The plugin works as follows: (1) it reads the generative process and loads the model with the selected subsystems; and (2) passes the result as input to each selected template, which is then processed by the plugin's template engine.

3.2.4 ProjectIT-Studio Plugins' Integration

The set of Eclipse.NET plugins that constitute ProjectIT-Studio are integrated via the import and extension point mechanisms provided by Eclipse.NET. Figure 3.4 illustrates the global plugin architecture of ProjectIT-Studio.

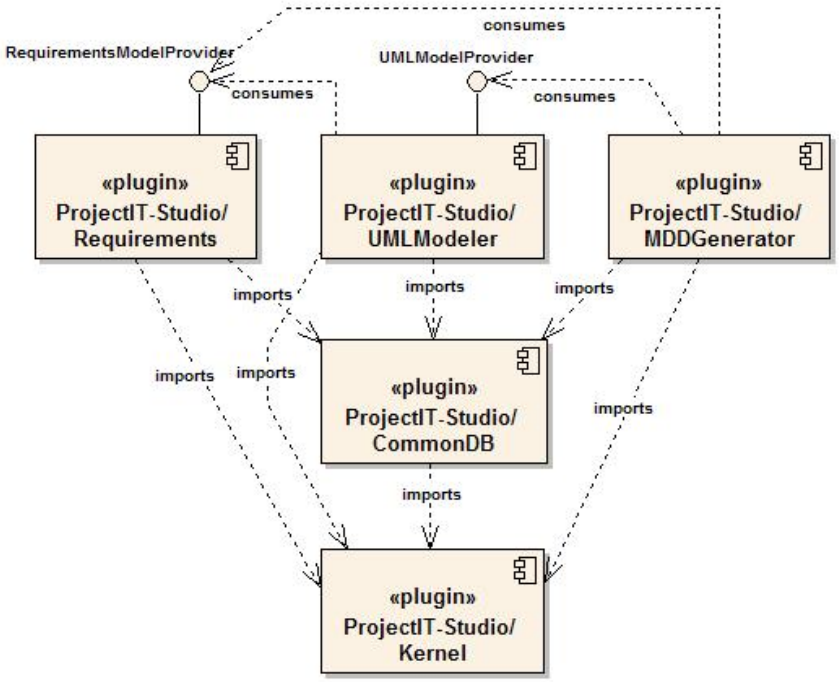

Figure 3.4: ProjectIT-Studio's plugins and relations between them (extracted from [Silva 06a]).

Each of the three main plugins of ProjectIT-Studio (Requirements, UMLModeler and MDDGenerator) are related among themselves only through extension points; this means that ProjectIT-Studio does not require that all three plugins are installed on the same system. Also, all the plugins that supply extension points are configured to present the user with a set of options according to the extensions provided for each extension point, allowing the choice of the consumer that should receive the model.

Requirements declares an extension point, called "RequirementsModelProvider", whose semantics declares that the plugin can provide UML models; any plugins that can process or consume UML models produced by Requirements should declare themselves as consumers of this extension point.

UMLModeler also declares an extension point, called "UMLModelProvider", with a purpose similar to the "RequirementsModelProvider" extension point. Additionally, since the ProjectIT approach includes obtaining models from requirements, UMLModeler declares itself as a consumer of the "RequirementsModelProvider" extension point.

MDDGenerator, as the last plugin to be used in the ProjectIT approach, does not declare any extension points, since there would be no consumers. The plugin does declare itself as a consumer of both UMLModeler and Requirements' extension points ("UMLModelProvider" and "RequirementsModelProvider", respectively). This way, MDDGenerator can receive models from both Requirements and UMLModeler, allowing the developer to: (1) directly generate code from the specified requirements; or (2) generate code from a model designed using the UML modeler.

Additionally, each of those three plugin requires the presence of some ProjectIT-Studio "base" plugins, namely the **ProjectIT-Studio/Kernel** and the **ProjectIT-Studio/CommonDB** plugins, whose function is to support the development-oriented top-level plugins. The Kernel plugin provides basic framework facilities, like the UML2 metamodel, to the top-level plugins. The CommonDB plugin allows the top-level plugins to serialize and deserialize information to a persistent storage mechanism (such as a relational database management system), decoupling persistence details from those plugins. Serialization to a persistence medium used by multiple ProjectIT-Studio instances allows those instances to be synchronized with each other.

3.3 ProjectIT-Studio and MDA

MDA certainly has the potential to adequately cover all the software development lifecycle phases more directly related to software itself, like implementation design (e.g., determining classes and interfaces) and coding. However, MDA does not address the requirements phase, leading to the known gap between "what the client wants/needs the system to do" and "what the system really does". This is partly because UML is sometimes not adequate for modeling non-software-related domains; nevertheless, the extended MDA framework does not specifically require the usage of UML, but a language that can be used to define the PIM and PSM languages, as Figure 3.5 illustrates.

ProjectIT-Studio is designed to make the ProjectIT approach (which is inspired by the extended MDA framework) an easy and efficient approach to use for software development.

Development of a project in ProjectIT-Studio involves the following steps: (1) specifying requirements using a DSL adapted to the "requirements specification" problem-domain; (2) transforming those requirements to an UML model – the PIM – with a specific Profile (usually XIS2); and (3) converting the PIM to source-code, by using a template mechanism. Note that, in the ProjectIT approach, PSMs are not used; instead, they are replaced by the generative templates.

Figure 3.6 illustrates how the ProjectIT-Studio plugins support the various steps of the ProjectIT approach.

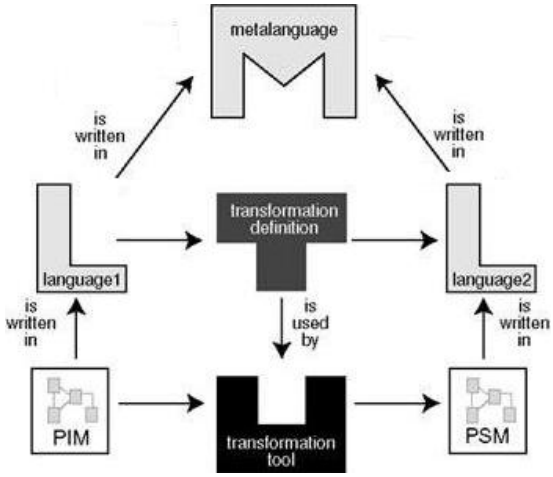

Figure 3.5: The extended MDA framework (extracted from [Kleppe 03]).

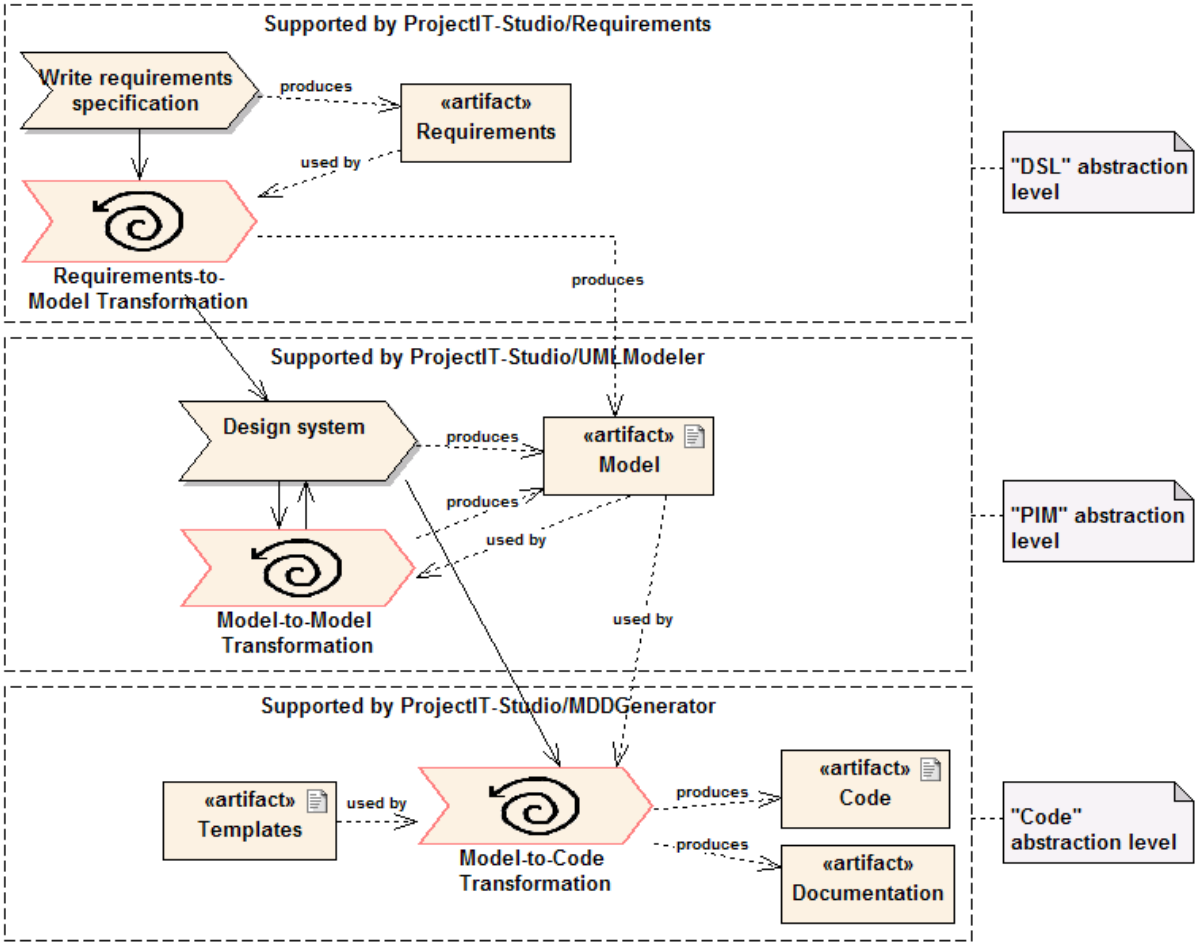

Figure 3.6: The ProjectIT approach, supported by the different plugins of ProjectIT-Studio.

Thus, the ProjectIT approach can be regarded as an instance of the extended MDA framework combined with a DSL that addresses requirements specification.

Chapter 4

The UML Modeling Plugin in ProjectIT-Studio

The UML modeling plugin, called **ProjectIT-Studio/UMLModeler**, is implemented as a plugin for Eclipse.NET. This plugin provides a graphical modeling environment to support the ProjectIT approach. In this environment, the Designer can create and refine UML models, with features such as model-to-model transformations, continuous "background" model validation, and UML Profile support. Additionally, UMLModeler and the other ProjectIT-Studio plugins use an UML metamodel implementation which conforms to the UML 2.0 [1] Superstructure specification [OMG 05b]; this UML implementation was also created in the scope of this work.

This chapter presents the following fundamental aspects of UMLModeler: (1) the UML metamodel implementation; (2) the functional architecture of UMLModeler; (3) the integration between UMLModeler and other plugins of ProjectIT-Studio; and (4) the support for external manipulation of the model.

4.1 The UML Metamodel Implementation

One of the most important components of UMLModeler (and of the other ProjectIT-Studio plugins), is the UML metamodel implementation, which is used in: (1) Requirements, to generate UML models at the end of the specification process; (2) UMLModeler, for most of the actions that the user can perform; and (3) MDDGenerator, to interpret UML models in order to generate artifacts. Thus, one of the most important stages of the development of UMLModeler was the analysis of the available UML metamodel implementations, and the decision of whether to use one of these implementation or de-

[1] For the remainder of this chapter, the term "UML" (with no number following it) will also be used, and it will always refer to UML 2.0 (unless explicitly stated otherwise).

velop a new one. This leads to the following basic requirements for the UML metamodel implementation: (1) methods should be efficient, so the ProjectIT-Studio user does not encounter noticeable performance drops; (2) the implementation should use the Observer design pattern [Gamma 95], so that the modeler will always be synchronized with the UML model being edited; and (3) its API should be simple to use and oriented towards the UML metamodel specification (i.e., without exposing implementation-specific parameters).

The adopted course of action was to initially use the Eclipse UML2 project's implementation [EclipseUML2] converted with the IKVM.NET tool [IKVM.NET], because this implementation accomplishes all of the previous requirements, except for the first (the implementation, when converted with IKVM.NET, introduces significant delays that are noticeable by the user, because of the reason previously explained in the "State of the Art", Chapter 2). The rationale for this decision was the following facts: (1) this implementation could be obtained quickly; (2) it was important to start implementing UMLModeler as soon as possible, in order to obtain as much feedback as possible on aspects such as desired functionalities and graphical look-and-feel; and (3) in the initial stages of development of the ProjectIT-Studio, the UMLModeler was the only plugin that needed this implementation, so an eventual metamodel replacement would not require extensive changes to all ProjectIT-Studio tools. It was also decided that when UMLModeler itself had reached a relatively stable stage, the metamodel implementation would be evaluated again (in terms of speed and easy-to-use API), and only then a final decision on the metamodel implementation to be used (either Eclipse UML2 or a new implementation, possibly with an API similar to the one exposed by Eclipse UML2) would be made.

This final evaluation of the Eclipse UML2 implementation leads to some important conclusions: (1) some parts of its API require some knowledge of the implementation's internal mechanisms; (2) there were relatively large delays when UMLModeler used the implementation's operations (such as adding or removing an Element from a Package); and (3) apparently, the IKVM.NET conversion was not totally correct, as there were several runtime exceptions that would be thrown with no apparent reason (e.g., the library could not be loaded when a debugger was currently attached to the process, and Elements could not be created within a package – they had to be created outside the package, and then added to it – even though the API supported these operations). Additionally, nUML [nUML] was still in an immature state, and it did not provide any way to detect changes to a model (such as by using the Observer design pattern). Thus, it was decided to create a new implementation of the UML metamodel that should accomplish all the previously mentioned requirements while following the UML Superstructure specification as best as possible.

The following subsections describe the fundamental aspects related to this new metamodel implementation (called **UMLModel**), the extensions that were made to the metamodel, and the issues that surfaced and how they were handled.

4.1.1 UML Superstructure Implementation

One of the first problems that surfaced during the implementation of UMLModel was that the UML Superstructure is defined using multiple-inheritance, but .NET-based programming languages (such as C#) only support single-inheritance. This was solved by using interfaces, since .NET-based languages do support a particular type of multiple-inheritance but only through the usage of interfaces [Archer 01]. The methodology for solving this problem was: (1) create an interface and a corresponding implementation class for each UML element; (2) establish inheritance relationships between those interfaces exactly as specified in the UML Superstructure specification [OMG 05b]; and (3) in those cases in which a child element inherits from two (or more) parent elements, choose the parent element which is most complex or provides the most functionality, and implement the functionality of the other remaining elements as interfaces, by copying code from their respective implementation classes. Figure 4.1 presents an example of this methodology: the `Package` interface inherits from the `PackageableElement` and the `Namespace` interfaces, and is implemented (or "realized", in UML terminology) by the `PackageImpl` class; on the other hand, the `PackageImpl` class only inherits from the `NamespaceImpl` class, but not from the `PackageableElementImpl` (because of the single-inheritance constraint imposed on classes), and all functionality from the `PackageableElementImpl` class must be manually copied to the child element's class. This methodology does have the disadvantage of requiring the manual copy of code between some classes, which is undoubtedly an error-prone task; however, this disadvantage can be somewhat minimized by making a careful and weighted choice of the parent element from which a child should inherit.

In order to handle the size and complexity of the UML Superstructure, it was decided to use an iterative and incremental approach to the development of UMLModel. Since the UML Superstructure is split into several packages, as illustrated in Figure 4.2, the first iteration of the implementation was focused on the `Classes` package, because it is the most fundamental package in the UML Superstructure; the other packages would be implemented in the following iterations. Additionally, this implementation was done by following the UML Superstructure specification as rigorously as possible, except for a small number of situations in which the textual description of the specification was ambiguous or even erroneous; in these cases, the implementation was done by using common-sense

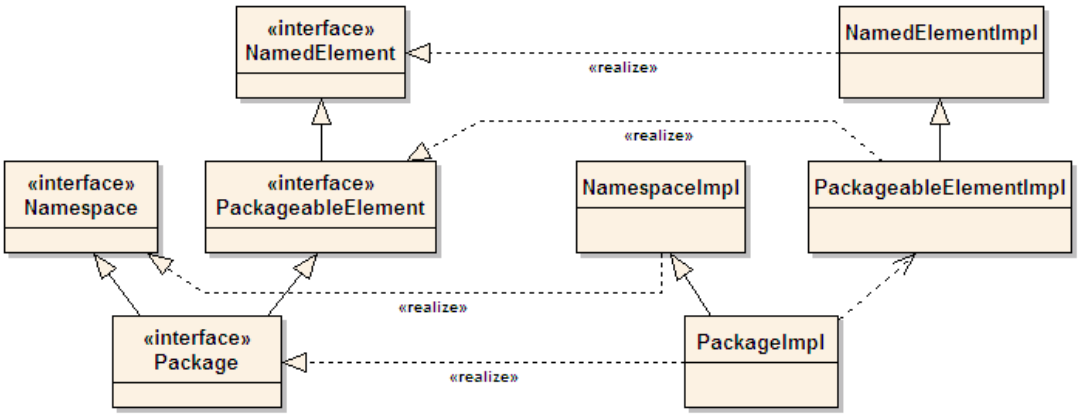

Figure 4.1: UML's multiple-inheritance reflected on a .NET-based language.

combined with the experience in UML modeling that the Information Systems Group has gathered over the last years.

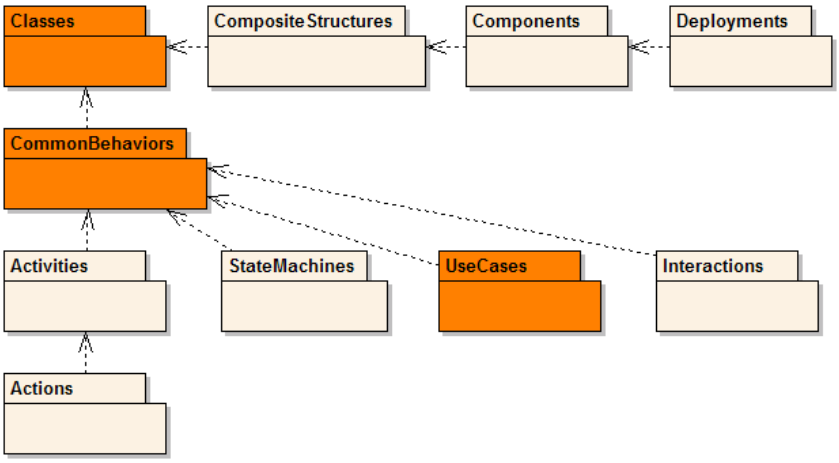

Figure 4.2: The packages that compose the UML Superstructure.

As Figure 4.2 shows, the `Classes`, `CommonBehaviors`, and `UseCases` packages have already been implemented in this work. All of the implemented packages are used by UMLModeler, which currently supports the following diagram types: (1) Class diagrams; (2) Package diagrams; and (3) UseCase diagrams. Nevertheless, UMLModeler will support the remaining diagram types in the future, as the remaining UML Superstructure packages are implemented in UMLModel and UMLModeler is adjusted accordingly.

4.1.2 Root Nodes and Views

Although the UML Superstructure specification does not define the concepts of *Root Node* and *View*, SparxSystems uses these concepts in Enterprise Architect [SparxSystems]. Furthermore, users apparently appreciated having these concepts available, as they allow

the division of a model into "smaller models", called **Root Nodes**, and the division of Root Nodes into "perspectives", called **Views** (which are actually Packages, although Enterprise Architect does not allow Views to be placed within elements other than Root Nodes). Figure 4.3 shows these concepts and the relationships between them.

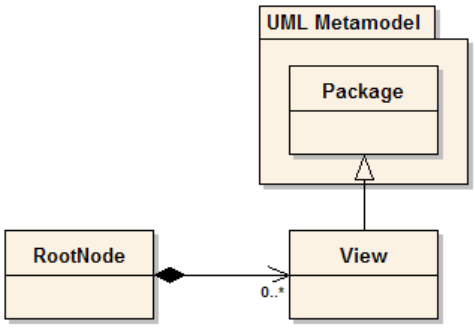

Figure 4.3: RootNodes and Views in Enterprise Architect.

It was decided that UMLModeler should also adopt these concepts, because they facilitate model management, and therefore `RootNode` was added to the UMLModel implementation. However, *View* was not added, since it consists of nothing more than a `Package` within a `RootNode`. Instead, it was decided that *UMLModeler would present `Packages` that are direct children of `RootNodes` as `Views`*, instead of creating a new View element with the single purpose of being a child of a `RootNode`. Figure 4.4 presents the Root Node in the context of UMLModel.

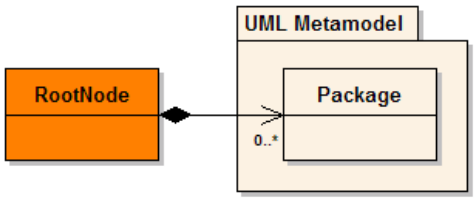

Figure 4.4: `RootNode` as a container of `Packages` (its `Views`).

4.1.3 Profiles and Stereotypes

The UML Profile mechanism [OMG 05b] is one of the most powerful aspects of UML, as it allows adding elements and semantics to UML. However, the definition of this mechanism in the UML Superstructure specification is more complicated than necessary, and lacks some essential elements (such as a precise definition of how to apply a Stereotype to an Element).

After a review of the new specification of the Profile mechanism, it was decided that a rigorous implementation of this mechanism in UMLModel would probably not be the

best approach; thus, some changes were made in order to make it easier for a model editor to manipulate a model and its applied profiles, and for the user to understand the model being manipulated. Figure 4.5 presents a structural overview of the Profile mechanism implemented in UMLModel.

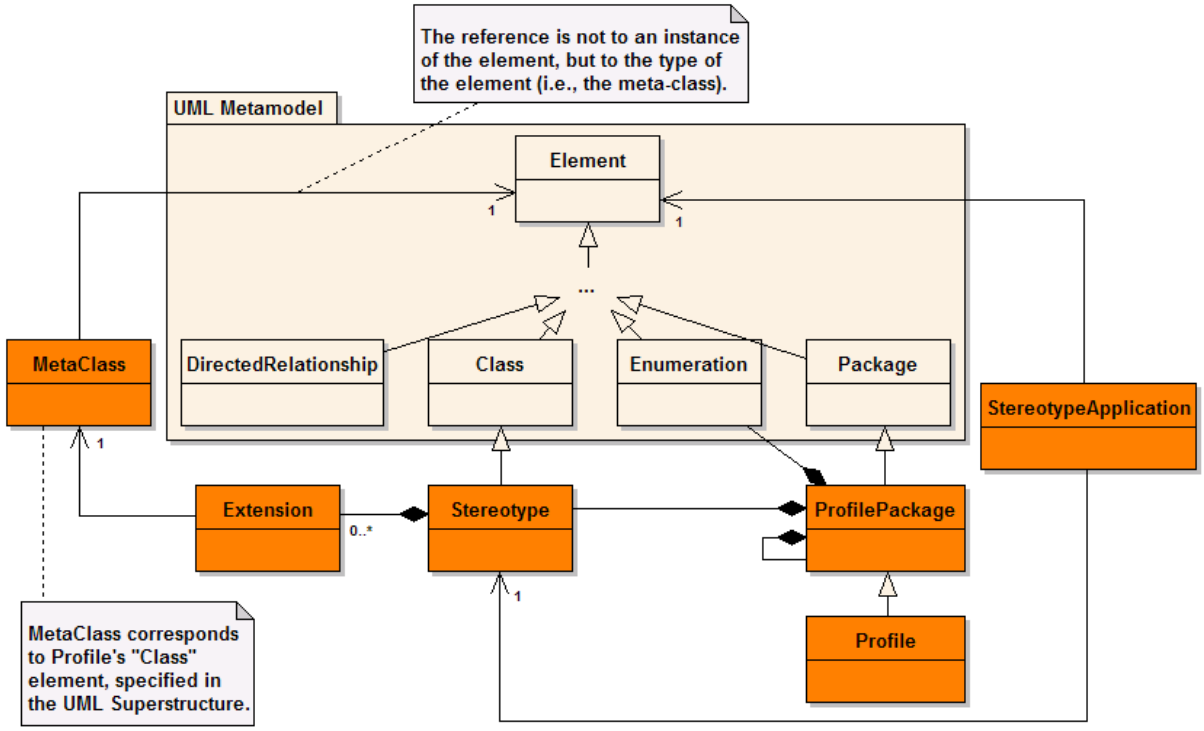

Figure 4.5: The Profile mechanism in UMLModel.

The Stereotype concept was implemented as a class, appropriately called `Stereotype`, that inherits from `Class` and follows the Stereotype definition in the UML Superstructure specification, except in the following aspects: (1) a class called `MetaClass` was added, being its only purpose to "wrap" the type of the UML Element being represented; (2) metaclass extensions are done through the `Extension` class, which now has a reference to a `MetaClass`; (3) extensions are stored in the `Stereotype` as a list of `Extensions` (instead of being stored in the extended `MetaClass`); and (4) the element `ExtensionEnd` (defined in the specification) was removed, because it did not provide any useful information, and no particular reason was found to justify its existence.

It should also be noted that `MetaClass` is actually the "Class" element defined in the "Profiles" package of the Superstructure. However, the name was changed in order to remove ambiguity (because of two elements with the same name: `Class` from the "Classes" package, and `Class` from the "Profiles" package) and to facilitate interpretation of the model.

The Profile itself was initially implemented as a class called `Profile` inheriting from `Package`, just as defined in the specification. One of the architectural decisions taken at the time was that a `Profile` would not contain other `Package`s; otherwise, a `Package` would need to be able to contain elements from profiles *and* "regular" models, thus potentially making an UML model harder to understand by the user, and possibly complicating the implementation of `Package`.

Nevertheless, one of the features requested by the users testing UMLModeler was exactly the ability to define packages within profiles, in order to allow the separation of a large number of `Stereotype` into "smaller" packages contained within the `Profile`; this request was particularly critical for defining the XIS2 Profile in UMLModeler, because of the large number of profiles and views that it defines [Silva 07].

This request was the reason for the introduction of `ProfilePackage`, which is another type of `Package`. `ProfilePackage` was implemented like a traditional `Package`, but UMLModeler treats `Package` and `ProfilePackage` differently (e.g., a `ProfilePackage` can only be a child of other `ProfilePackage`s, and `Package` can only be a child of other `Package`s), thus allowing the user to make a clear separation between the profile and the UML model. `Profile` was then defined as a `ProfilePackage`, but with the added restriction that a `Profile` can not contain other `Profile`s.

Finally, the UML Superstructure specification defines an element, "*ProfileApplication*", that allows the application of a Profile to a Package. However, the specification does not define a similar element for the application of a Stereotype to an UML Element; towards this end, UMLModel introduces `StereotypeApplication`, which establishes a mapping between an Element and a Stereotype, thus allowing UMLModeler to apply a Stereotype to an Element. It should be noted that `StereotypeApplication` only stores the *fully-qualified name* [OMG 05b] of the `Stereotype`, instead of directly referencing it; this allows the user to temporarily remove a profile application and later reapply that same profile, without losing any information about the applied `Stereotype`s (such as tagged values).

"ProfileApplication" was not implemented in UMLModel, because the semantics of the UML model class defined by UMLModel (`ModelContents`, presented further down this chapter) provide an implicit application of Profiles to Packages.

The reading of the "Profiles" package specification in [OMG 05b] is highly recommended, as it provides a greater insight into the Profile mechanism.

4.1.4 Visual Representation

Another aspect that had to be handled was the visual representation of UML elements. The UML metamodel itself is only capable of containing *conceptual* information about models, but not *visual* information; although the metamodel specification does indicate how elements are visually represented, the elements themselves do not hold information regarding their visual representation.

This issue was handled in the initial stage of the development of UMLModeler by extending the UML metamodel to include some new concepts, which act as visual "proxies" [Gamma 95] to UML elements. The concepts defined in this extension are the following: (1) `ViewElement`, an abstract class which provides basic functionality and implements the Observer pattern; (2) `ViewNode`, an abstract class which represents a *node* in a diagram; (3) `ViewConnection`, an abstract class which represents a *connection* between two nodes – its source and its target – in a diagram; and (4) `Diagram`, which is basically a container of `ViewNode`s and `ViewConnection`s. Figure 4.6 presents the concepts defined in this work for visually representing UML elements, and how they are related to the represented elements themselves. It should be noted that these new elements do not require changing any of the UML elements defined in the UML Superstructure specification, so any implementation of the UML metamodel could be used (such as Eclipse UML2).

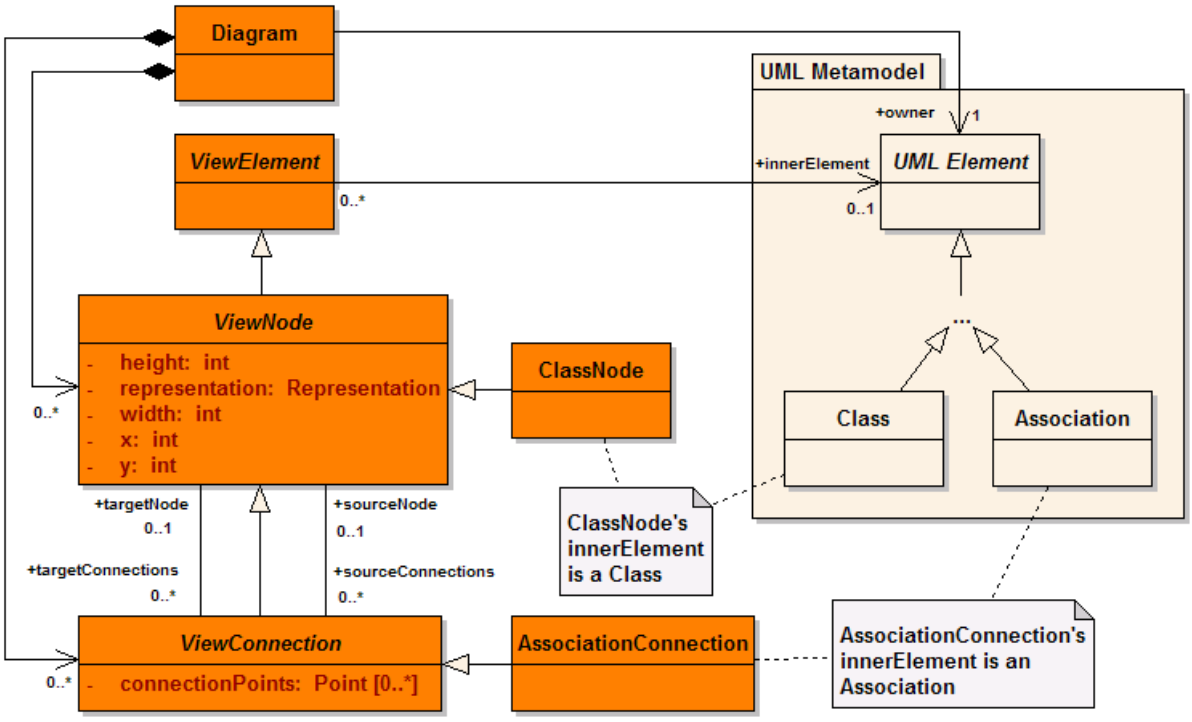

Figure 4.6: UML elements are represented by ViewElements in Diagrams.

The OMG had also recognized that this issue made it difficult to exchange visual models between tools, which is why the Diagram Interchange 1.0 specification [OMG 06b] was released in April of 2006 (halfway through the development of UMLModeler). The specification also consists of an extension to the UML metamodel, and presents concepts very similar to those introduced in this work; this means that the UML models handled by UMLModeler have a relatively high degree of alignment with the OMG specifications (regarding the conceptual and visual information of models), which is certainly an advantage of UMLModeler over other modeling tools.

A `ViewElement` has a reference to an UML element, its `innerElement`, for which it provides a visual representation (e.g., a `ClassNode`, which inherits from `ViewNode`, provides a visual representation for an UML `Class`). Similarly, a `Diagram` also contains a reference to an UML element, its `owner`; however, unlike what happens with `ViewElement`, a `Diagram` does not graphically represent an element, but is "owned" by it.

A `Diagram` contains separate lists for `ViewNode`s and `ViewConnection`s (instead of a single list of `ViewElement`s) in order to facilitate the set of operations that UMLModeler must support; these lists are implemented as C# generic lists (i.e., `List<ViewNode>` and `List<ViewConnection>`), so there is a greater level of type-checking at compile-time (because type-casting operations are not necessary), which helps prevent many errors that would otherwise be detected only at runtime.

The reason why `ViewConnection` inherits from `ViewNode` instead of inheriting from `ViewElement` is to allow a `ViewConnection` to act as `ViewNode`; this enables the possibility of having a `ViewConnection` connected to other `ViewConnection`s, as Figure 4.7 illustrates. Of course, this situation could also have been modeled as a `ViewConnection` having `ViewElement`s as its source and target; however, the first option also presents the advantage of reducing the number of required type-casting operations, which provides the benefit of better type-checking at compile-time.

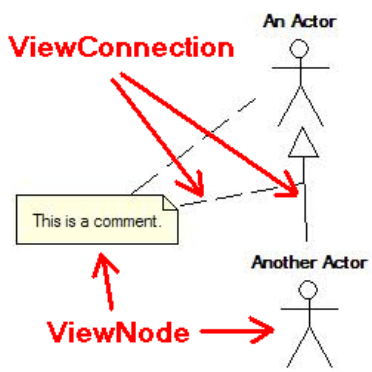

Figure 4.7: A `ViewConnection` can be connected to other `ViewConnection`s.

4.1.5 Serialization

Serialization is handled by the **Serialization package**, which provides the ability to *serialize all information about an UML model (including diagrams and stereotype applications) to a single XML file.* The decision to save the entire model to a single XML file was based on the following rationale: (1) XML is a format suitable for the exchange of information between different systems; (2) the XMI standard, which is traditionally used for exchanging UML models, is based on XML; and (3) saving all information about a model into a single file facilitates the exchange of models between different instances of ProjectIT-Studio (a model can be used on another instance by simply copying the corresponding XML file).

This package was defined in the initial stage of the development of UMLModeler, because of two reasons: (1) one of the problems that Eclipse UML2 presented (when converted with IKVM.NET) was that the invocation to serialize a model into XMI threw a runtime exception; and (2) the extension created in this work, to support visual information about a model, was obviously not supported by the Eclipse UML2 serialization mechanism.

One of the first and most important decisions in the creation of this package was *where serialization functionality should be defined.* There were two alternatives, each with its advantages and disadvantages: (1) implementing the serialization functionality within each UML element; or (2) implementing the serialization functionality outside of the UML elements.

With the first alternative, each UML element should provide an additional method that returns a XML element containing all information about that element and the elements contained within it. This alternative presented the following advantages: (1) changes to an UML element's implementation would not need to be propagated to another class; and (2) traversal of the model could be easily accomplished, by using the **Visitor design pattern** [Gamma 95]. On the other hand, it also presented some disadvantages: (1) added complexity to the implementation of each UML element; (2) the need to handle object references between elements would require that each element should be able to access the entire model; and (3) it required changing the code of the UML metamodel implementation.

The second alternative also presented some benefits and issues, most of which were the opposite of the advantages and disadvantages of the first alternative. The advantages were: (1) no added complexity to the implementation of each UML element; (2) since the entire model was available, it would be much easier to handle object references; (3) the model could be traversed by using the **Two-Step View design pattern** [Fowler 03]; and (4) any metamodel implementation (including Eclipse UML2) could be used. Likewise,

its disadvantages were: (1) any changes to UML elements would need to be reflected in an external class; and (2) model traversal logic would be defined outside of the elements being traversed.

The first alternative would require changing the code of the Eclipse UML2 implementation, so initially the second alternative was chosen. When the decision was made to create a new UML metamodel implementation, these alternatives were again considered, as either alternative could be easily implemented; however, after careful consideration, it was decided that it would be preferable that the Serialization package was totally defined outside of the UML metamodel implementation itself (in order to maintain their functionalities relatively independent), and so the second alternative remained in effect.

To facilitate its design and implementation, the Serialization package was specified to handle two smaller aspects: (1) save (or *serialize*) the model to XML, and (2) load (or *deserialize*) the model from XML. The only common point between these aspects is the set of strings used, such as the names of XML nodes and attributes, or possible values for an XML node; this common point was implemented by an abstract class, `SerializationBase`, that provides all the strings required for storing/loading a model. Figure 4.8 presents the structure of the Serialization package.

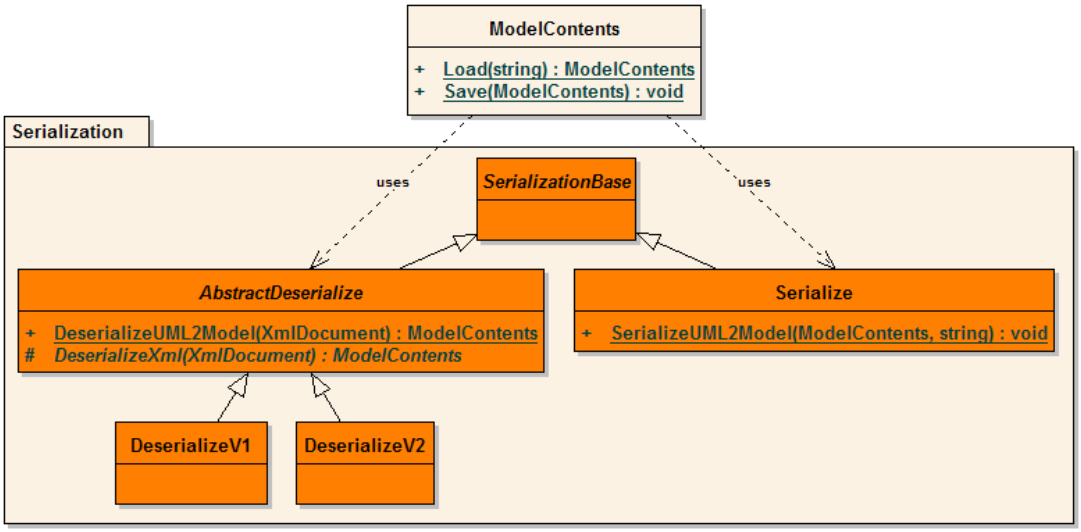

Figure 4.8: The structure of the Serialization package.

The *serialization* aspect was handled in a straightforward manner, by using the Document Object Model (DOM) [XML b] standard for handling XML documents. It was implemented by a simple class, called `Serialize`, which provides a single public method that receives an UML model and a string indicating the path of the file where the model should be stored. This method does the following actions: (1) receive the model and the

file path; (2) build a DOM-based XML document while traversing the model; and (3) after the model traversal is complete, save the XML document to a file in the given path.

The *deserialization* aspect was initially handled in that same way, by parsing a DOM-based XML document and constructing the corresponding UML model. Nevertheless, this approach presented an additional issue, because it became necessary (halfway through the development of UMLModeler) to make some changes to the format of the XML document generated by `Serialize` in order to better align the document with the implementation. This change presented a problem, as UMLModeler was already being used to create models: if the deserializer was changed in order to reflect the new document's format, then UMLModeler would not be able to load models saved with the previous format. This problem was handled by using the **Strategy design pattern** [Gamma 95], and required: (1) the addition of an attribute to the root node of the XML document, specifying the *version* of the serialization format used to save that UML model; and (2) the addition of the abstract class `AbstractDeserialize`, which provides a single static method that receives a XML document from where the model should be loaded, and returns the corresponding UML model. This method performs the following actions: (1) find the version of the model that was saved, and get the appropriate deserializer (using the Strategy design pattern); and (2) provide the XML document to the deserializer and return the model returned by the deserializer. The classes `DeserializeV1` and `DeserializeV2` are the deserializers currently used by UMLModeler; any changes to the format of the XML document generated by `Serialize` would only require the creation of a new deserializer that correctly interprets the new format.

The Serialization package is not yet capable of serializing in the XMI format, because the format currently used is oriented towards the implementation, which facilitates the detection and debugging of potential problems with saving and restoring information about a model. However, *the current implementation of this package could easily be modified to support XMI*, as it would likely involve only changing the strings used, some adjustments to the DOM-based XML document, and creating a new deserializer (e.g., `DeserializeV3`) that can correctly handle XMI; this modification is planned to be made in the very near future.

4.1.6 ModelContents

Finally, it was necessary to define a class that represents a complete UML model and aggregates all the model information previously presented. This class is called `ModelContents` and is presented in Figure 4.9, along with the relationship between this class and all the model information presented in the previous subsections.

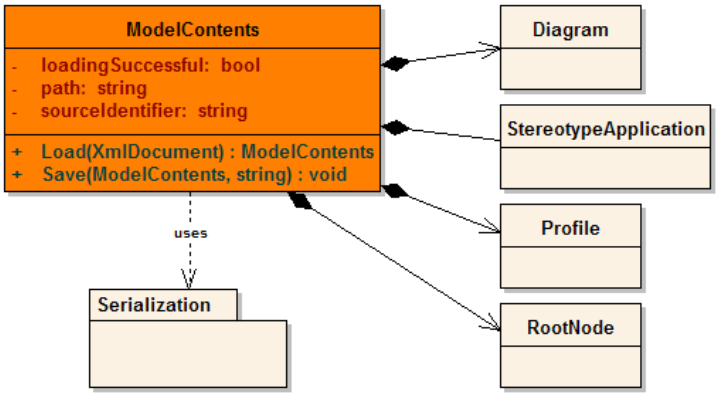

Figure 4.9: The relationship between `ModelContents` and all model information.

`ModelContents` acts as a container for `Diagrams`, `Profiles`, `StereotypeApplications`, and `RootNodes`. Besides acting as a container for those elements, `ModelContents` contains some additional information that can be used for any number of purposes: (1) `loadingSuccessful` is a boolean flag that indicates whether deserialization from a XML file was successful; (2) `path` is a string that indicates the path of the file where the model has been serialized (or the file from which the model was deserialized); and (3) `sourceIdentifier` is a string used by UMLModeler to establish a mapping between an identifier and a file path to where the model should be serialized (this mapping is used in the integration between UMLModeler and Requirements).

The `ModelContents` class also provides additional functionality (in the form of utility methods) that can be used by UMLModeler and by any other plugin that wishes to query or manipulate an UML model. Examples of this functionality are: (1) methods to set and get the default diagram for each element; (2) two static methods, `Save` and `Load`, which are used to invoke `Serialize` and `AbstractDeserialize` functionality, respectively; (3) a method to get all `Stereotypes` applied to an element; and (4) a method to get a `Stereotype` given its fully-qualified name. The complete listing of these utility methods is out of the scope of this chapter, and is available in "Appendix C – ProjectIT-Studio Programmer's Manual".

`ModelContents` also provides the semantics that any `Profiles` contained within it are automatically applied to all `Packages` within its `RootNodes`. This is the reason why the "ProfileApplication" element was not implemented, as the presence of a `Profile` within a `ModelContents` already implies the presence of `ProfileApplications` between that `Profile` and all the other `Packages` contained in the `ModelContents`.

4.2 Functional Architecture

UMLModeler was developed using an approach based on packages, each of which provides a specific set of functionalities within ProjectIT-Studio. An example of these packages is shown in Figure 4.10, which presents a typical screenshot of UMLModeler, highlighting its main packages (with which the user will interact more often): the *Graphical Modeler* and the *Content Outline*.

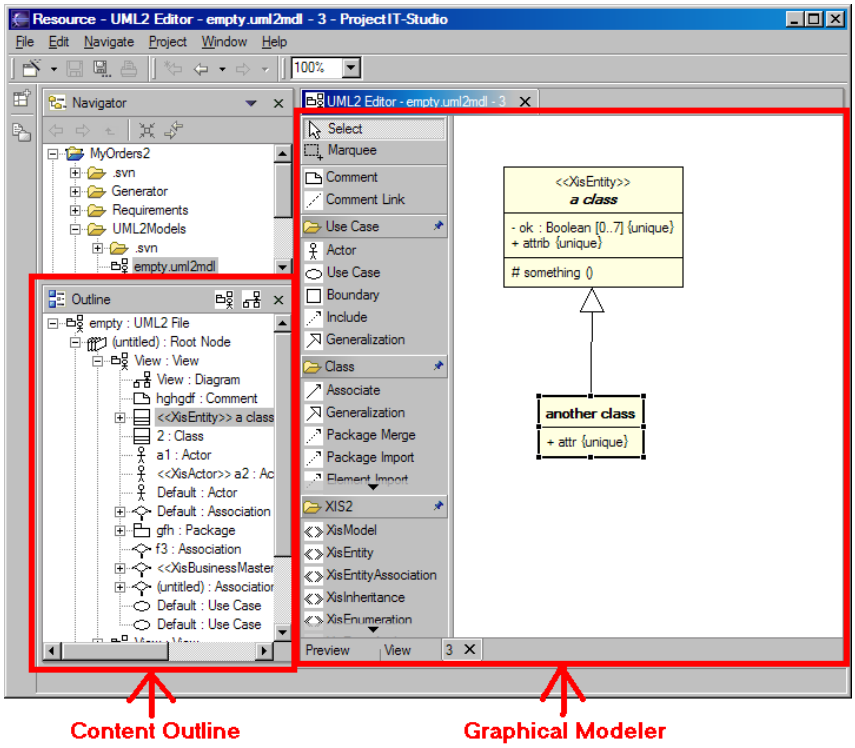

Figure 4.10: Screenshot of UMLModeler, highlighting its main packages.

Besides these main packages, UMLModeler also contains a number of auxiliary packages, which handle issues like: (1) setting UMLModeler's configuration; and (2) displaying windows that contain all the information about a certain UML element.

Figure 4.11 presents an overview of the packages of UMLModeler's architecture, which are described in the next subsections.

4.2.1 Content Outline and the Outline Page

The **Content Outline** is a mechanism provided by Eclipse.NET for displaying information about the document currently being edited. Any editor plugin can use this mechanism, by simply providing an **Outline Page** that should present a high-level view of the corresponding document.

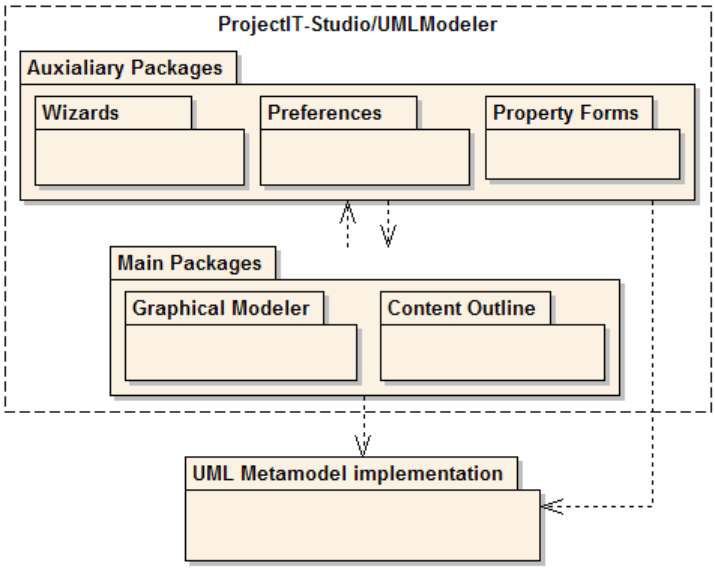

Figure 4.11: The packages of UMLModeler's functional architecture.

UMLModeler provides such an Outline Page, which is shown in Figure 4.10 and Figure 4.12. This Outline Page presents a tree that reflects the hierarchical structure of the UML model currently being edited. The root of this tree corresponds to the `ModelContents` that represent the UML file itself, and its children are the Root Nodes and the Profiles defined within that `ModelContents`. Any stereotype applications to an UML element are shown as the stereotype's name between guillemots (e.g., <<Stereotype name>>) before the element's name. A Diagram is shown as a child of the UML element that owns it.

The Outline Page also provides a set of possible actions that can be performed over the model. The available actions are presented through the Outline Page's context menu (accessible by a click of the right mouse button over a tree item), which only presents the set of actions that make sense in the context of the selected item (e.g., right-clicking over an UML class will not present an action to create a Profile).

Figure 4.13 presents an overview of the classes that allow UMLModeler to provide its Content Outline functionality.

The Outline Page itself is implemented by the class `OutlinePage`, which is supported by the `LabelProvider` and the `ContentProvider` classes. The `LabelProvider` provides the image and the strings that are displayed by each tree item; on the other hand, the `ContentProvider` is responsible for determining the children of each item (which is necessary for expanding/collapsing tree items), and for "observing" the model and updating the tree items if the model changes. Additionally, `OutlinePage` interprets mouse double-clicks as a request to open the selected element's Property Form, and thus forwards this request to the `PropertyFormFrontController` (described in the next subsection).

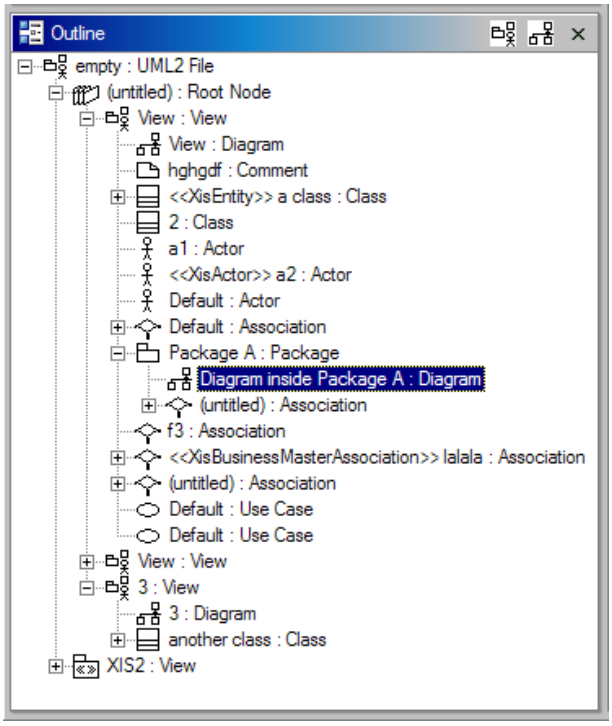

Figure 4.12: Screenshot of the Outline Page provided by UMLModeler.

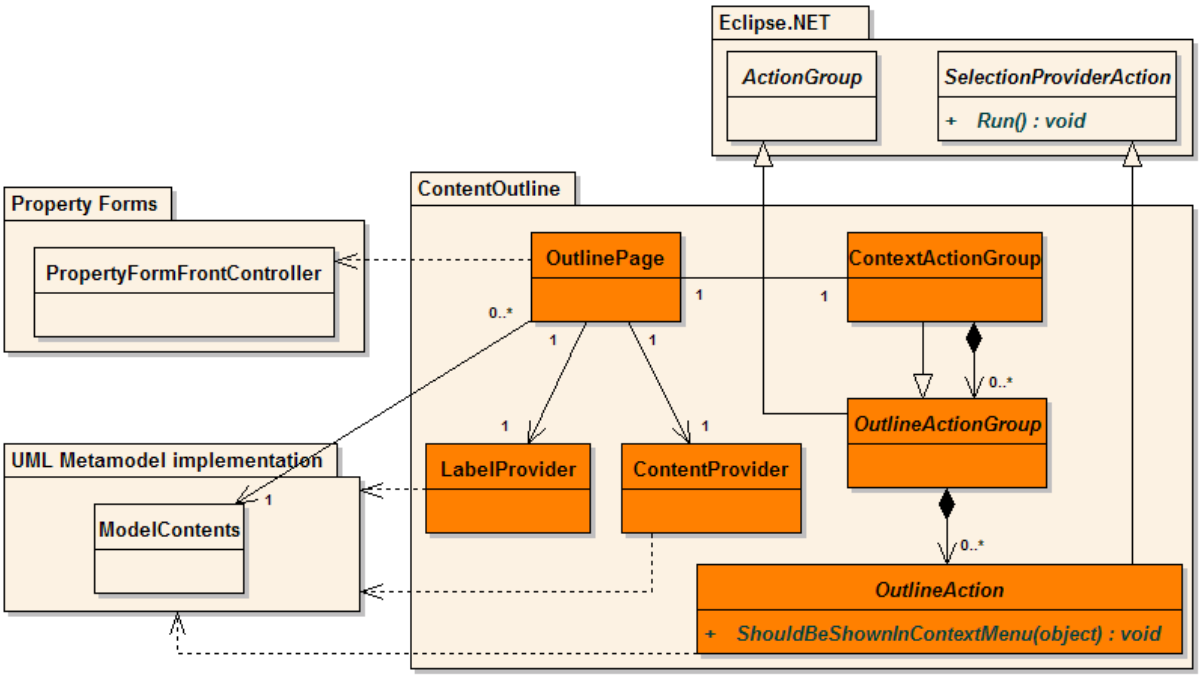

Figure 4.13: The classes that provide Content Outline functionality.

The context menu actions are provided through the classes `ContextActionGroup`, `OutlineActionGroup`, and `OutlineAction`. The class `OutlineAction`, which inherits from the abstract class `SelectionProviderAction` provided by Eclipse.NET, represents

an action presented to the user (as a menu item in the context menu), and there are no architectural restrictions on what each of those `OutlineActions` may perform; this class features a method, `ShouldBeShownInContextMenu`, which receives the object selected by the user, and determines whether that `OutlineAction` should be available in the current context menu. The class `OutlineActionGroup` inherits from the abstract class `ActionGroup` provided by Eclipse.NET, and consists of nothing more than a logical grouping of `OutlineActions` (e.g., a grouping of actions to edit an element, or a grouping of actions to create additional child elements). The `ContextActionGroup`, which inherits from `OutlineActionGroup`, acts as a singleton that contains other `OutlineActionGroups`, and is the only class with which the Outline Page directly interacts.

Additionally, the Outline Page also supports *drag-and-drop operations* between its elements. This functionality allows the user to change the hierarchical structure of the UML model currently being edited (e.g., a user can change the owner of a `Diagram` simply by dragging it to its new owner, or change the owning `Package` of a `Class` by dragging the `Class` from its previous owner to the new owner). The Outline Page immediately validates these drag-and-drop operations, and provides visual feedback indicating whether the operation is allowed.

It should be noted that this hierarchical structure will likely suffer some changes when the integration with ProjectIT-CommonDB is made. However, this change should be confined to the `ContentProvider` class, and possibly to some `Actions`, because the other classes are completely independent of the source of the model being edited.

4.2.2 Graphical Modeler

The Graphical Modeler was implemented by using a modified version of the Eclipse Graphical Editing Framework (GEF). The modifications made were the following: (1) the source code (written in Java) was changed, so that it would use the .NET Framework as much as possible by using *ikvmstub* (e.g., usages of `java.util.ArrayList` were replaced by `System.Collections.ArrayList`); and (2) after changing the source code and compiling it, the resulting .JAR files were converted to .NET assemblies, by using *ikvmc*.

The Graphical Modeler itself is divided into several packages, each of which provides a set of functionalities that serves the following purposes: (1) define a specialized configuration of the framework provided by GEF; and (2) adjust UMLModeler to facilitate modeling tasks in the context of the ProjectIT approach.

Figure 4.14 presents an overview of the packages that define the Graphical Modeler and all its functionality.

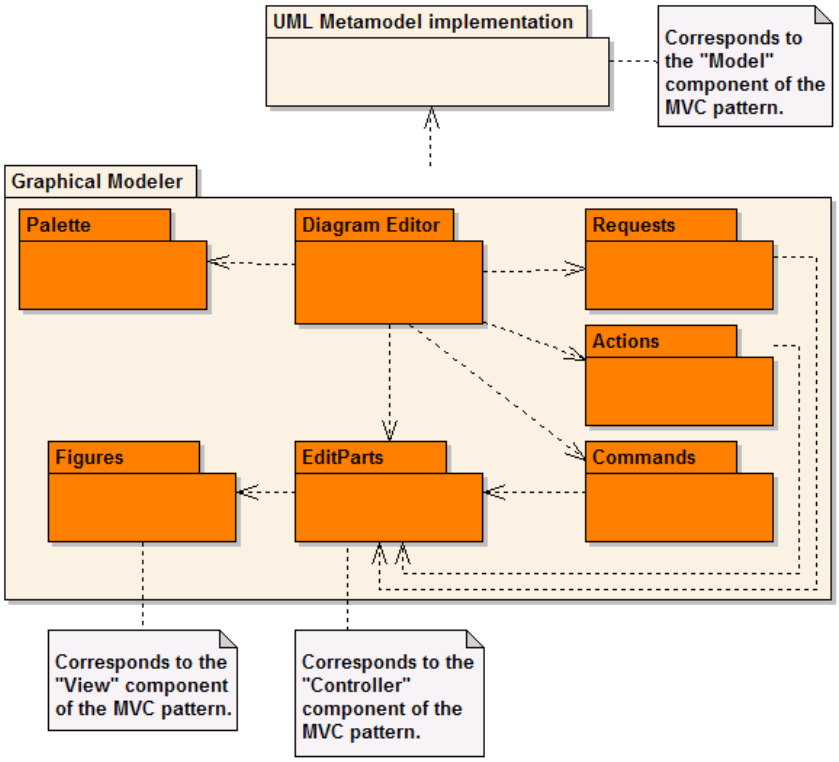

Figure 4.14: The functional architecture of the Graphical Modeler.

Diagram Editor

The **Diagram Editor package** is the main package of the Graphical Modeler, and it contains only one class, `UMLDiagramEditor`. This class inherits from the abstract class `GraphicalEditorWithPalette` provided by GEF, which already handles most of the aspects regarding the creation and maintenance of a graphical editor and its palette (illustrated in Figure 4.15).

`UMLDiagramEditor` is responsible for: (1) initializing and invoking all the other packages in the Graphical Modeler; (2) providing the diagram being edited (or even the `ModelContents` itself, if necessary) to any class which requires it; and (3) interpreting life-cycle events (e.g., a "Save" request) sent by either GEF, Eclipse.NET, or the user.

EditParts

The **EditParts package** contains the EditParts that are used by UMLModeler as the *controllers* of the Graphical Modeler, which is based on GEF. These EditParts are responsible for most of the functional aspects of the Graphical Modeler, such as: (1) interpreting the user's input, such as a double-click over the representation of an UML element, or the dragging of a representation from one location to another; and (2) detecting changes to their corresponding *model* and updating their visual representation accordingly.

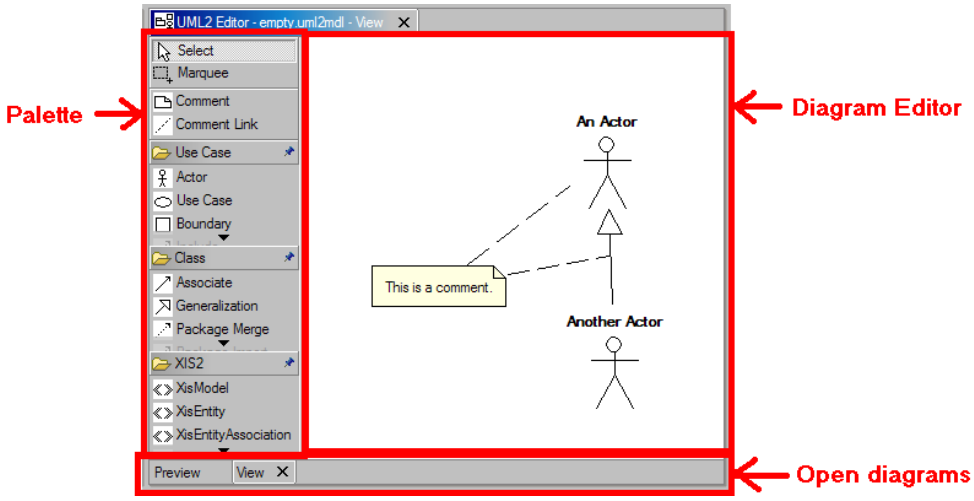

Figure 4.15: Screenshot of the Graphical Modeler, highlighting its main visual components.

The class `UMLEditPartFactory` contains the `CreateEditPart` method that receives a *model* (i.e., an object from UMLModel, such as a `ViewNode` or a `ViewConnection`) and returns a new instance of the corresponding *controller* (i.e., an `EditPart`). This is the first class in this package with which UMLModeler interacts.

The top-level EditPart (i.e., the one that has no parent) is the `UMLDiagramEditPart`, which is the controller for a `Diagram`, and the parent to all the EditParts that correspond to the `ViewNodes` and `ViewConnections` contained in the `Diagram`. This package also defines the following base classes: (1) `UMLNodeEditPart`, which is the base class for all EditParts that are controllers for `ViewNodes`; (2) `UMLConnectionEditPart`, which is the base class for all EditParts that are controllers for `ViewConnections`; and (3) `UMLParentedElementNodeEditPart`, which is the base class for all EditParts that are controllers for other elements which are "contained" within `UMLNodeEditParts` or `UMLConnectionEditParts` (typically UML elements which are represented using a textual notation, such as `Attributes` and `Operations`). Figure 4.16 presents a structural overview of the essential classes of this package.

The top part of Figure 4.17 shows an example of the class instances involved in a diagram that contains three elements: (1) a `Class`, (2) a `Comment`, and (3) a connection between the Class and the Comment (indicating that the Comment is applied to the Class). The "UML Elements" section shows the UML elements involved in this example; note that there is no element representing the connection between the Comment and the Class, because UML has no element to represent such a connection. The "View Elements" section shows the corresponding diagram elements, now including an element to represent the connection between the Comment and the Class. Finally, the "EditParts" section

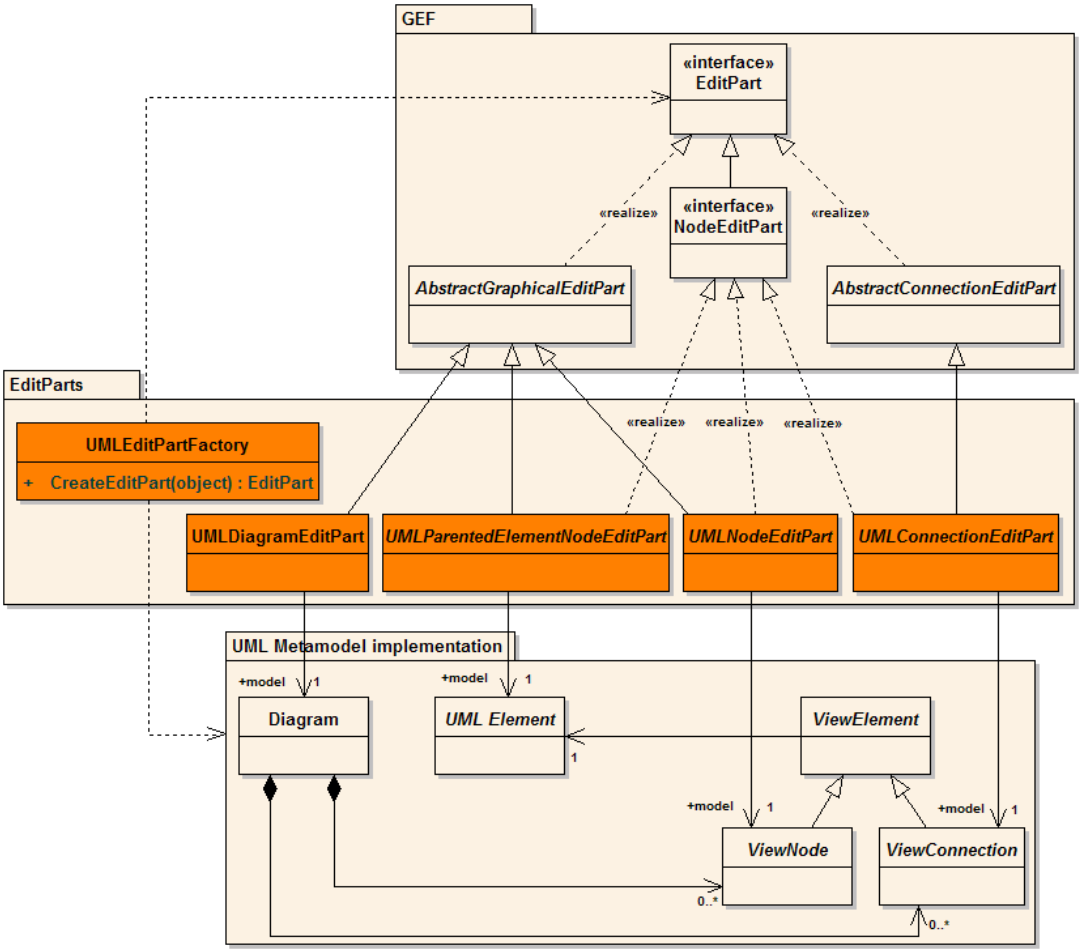

Figure 4.16: Overview of the classes of the EditParts package.

shows the corresponding EditParts. The bottom part of the figure shows those three elements in the Graphical Modeler. This example also shows that *the "UML element-to-view element" mapping is not strict*, as there are some View Elements that have no corresponding UML element.

Figures

The **Figures package** consists of all the figures used by UMLModeler as the *views* of the Graphical Modeler. These figures are responsible for presenting a complete visual representation of their corresponding *models*, according to the Model-View-Controller pattern. Figures usually contain a set of smaller figures, each of which presents a specific aspect of the corresponding model (e.g., the figure that represents an UML Class contains figures for each of the Attributes and Operations of the Class).

This package defines the following types of `Figures`: (1) `UMLNodeFigure`, which is the base class for all figures that define the visual representation of `ViewNodes`; (2)

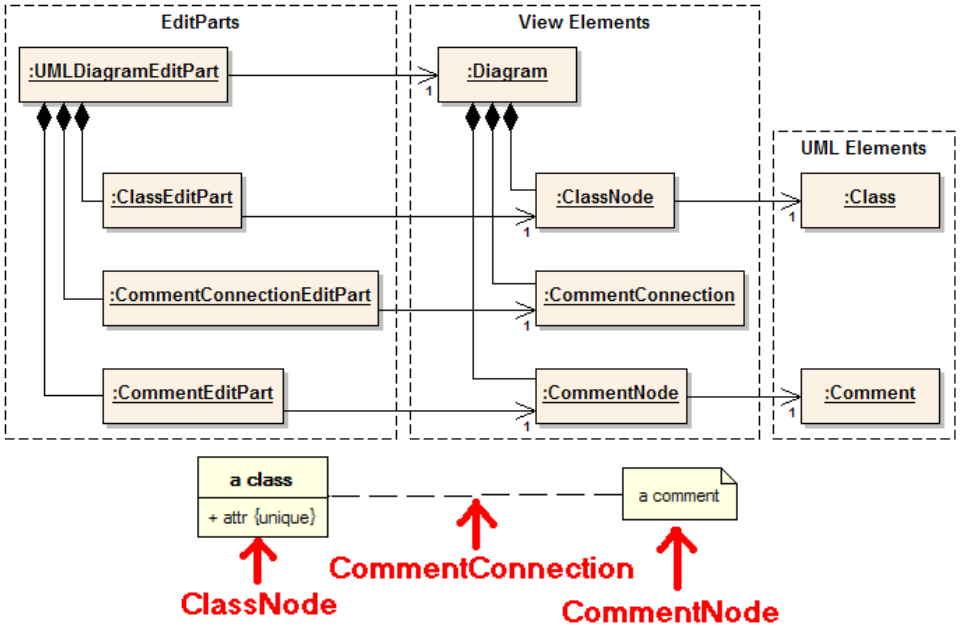

Figure 4.17: Class instances involved in a connection between a Comment and a Class.

`UMLConnectionFigure`, which is the base class for all figures that define the visual representation of `ViewConnections`; and (3) `UMLTextNodeFigure`, which is the base class for all figures that define the representation of UML elements represented by textual notation. All of these base classes inherit from classes defined by Draw2D, which provides visual representation functionalities for GEF. Figure 4.18 presents a structural overview of the essential classes of this package.

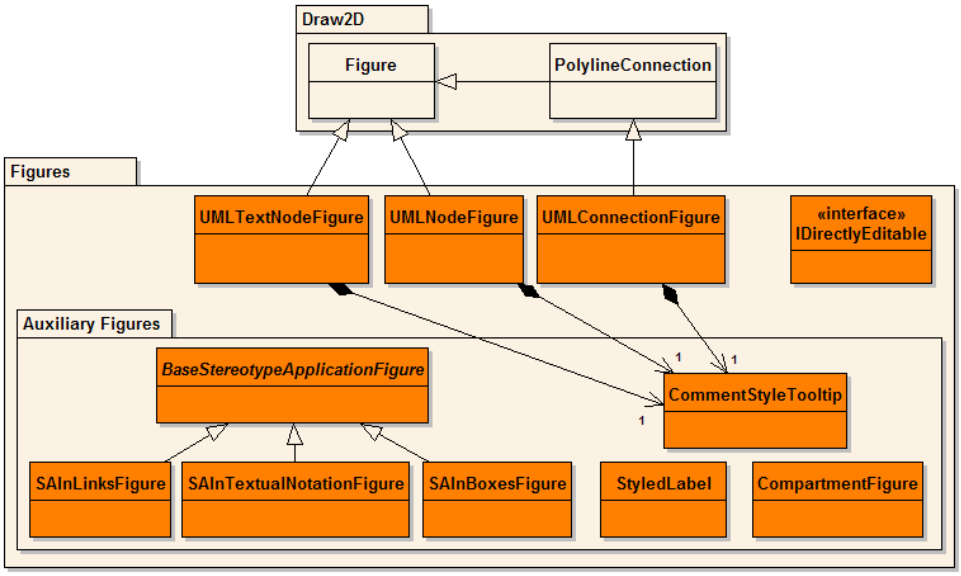

Figure 4.18: Overview of the classes of the Figures package.

Typically, there is a mapping between the `EditParts` (defined in the EditParts package) and the `Figures` defined in this package: (1) a `UMLNodeEditPart` usually has a `UMLNodeFigure`; (2) a `UMLConnectionEditPart` usually has a `UMLConnectionFigure`; and (3) a `UMLParentedElementNodeEditPart` usually has a `UMLTextNodeFigure`. Nevertheless, this mapping is not imposed by either UMLModeler or GEF.

Additionally, this package defines an interface called `IDirectlyEditable`, which must be implemented by all figures that allow the *direct edit* of their contents. **Direct Edit** is a functionality provided by GEF, which allows the user to perform the following sequence of actions: (1) press the "F2" key or single-click the mouse over the figure; (2) insert or edit the information in an SWT widget that appears over the figure (usually a text box); and (3) press the "Enter" key or click somewhere in the diagram, in order to stop this operation and update the model and figure with the information introduced by the user. A typical example of this functionality is the quick editing of an element's name.

The Figures package also contains a child package, `Auxiliary Figures`, with the following figures: (1) `CommentTooltipFigure`, which is used to display information about an element (such as Stereotype applications and corresponding tagged values) in a tooltip that looks like a Comment; (2) `StyledLabel`, which consists of a figure that displays text with a certain font format (such as bold or italic); (3) `CompartmentFigure`, which consists of a figure that facilitates the grouping of child figures (as an example, this figure is used to present the Attributes and Operations compartments in the Class figure); and (4) `BaseStereotypeApplicationFigure` and its child figures (`SAInLinksFigure`, `SAInBoxesFigure`, and `SAInTextualNotationFigure`), which are used to display stereotype applications in `UMLConnectionFigure`s, `UMLNodeFigure`s, and `UMLTextNodeFigure`s, respectively (by using either textual notation or images, according to the UML Superstructure specification). These figures also update themselves automatically when the user changes the UMLModeler preference options (in the Preferences dialog, explained further down this chapter), which removes some complexity from the classes of the Figures package.

Palette

The **Palette package** defines the contents of the Graphical Modeler's **palette**, which is shown on the left side of Figure 4.15. Figure 4.19 presents an overview of the Palette package.

The main class of this package is the `UMLModelerPalette` class, which represents a palette that uses the *palette drawers* and *palette tools* provided by GEF (which are shown in Figure 4.20). The Graphical Modeler palette contains several *drawers* (typically one for each type of UML diagram), each of which contains several *tools* (one for each available

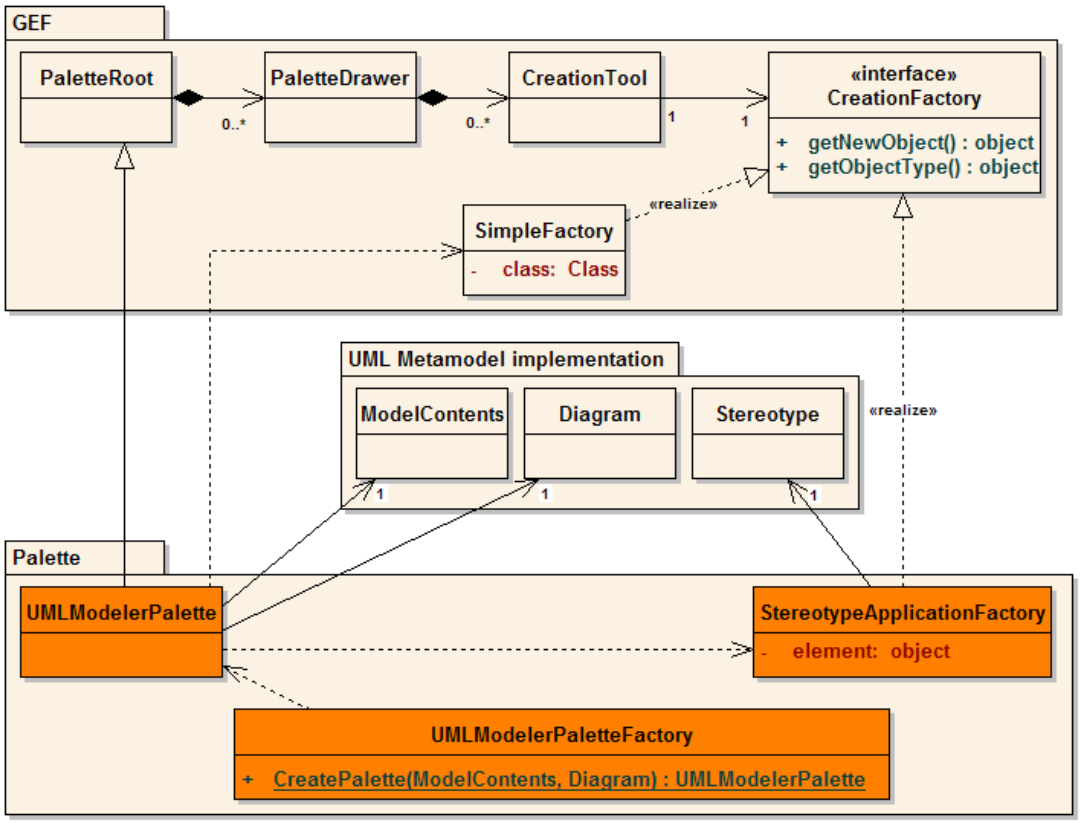

Figure 4.19: Overview of the classes that provide palette-related functionality.

`ViewNode` and `ViewConnection`). These tools allow the user to create new UML elements in the diagram, by clicking on the desired tool and: (1) if the tool corresponds to a `ViewNode`, clicking over the desired location for the node in the diagram; or (2) if the tool corresponds to a `ViewConnection`, clicking first over the *source* node and afterwards over the *destination* node. Note that a `ViewConnection` is also a `ViewNode`, so the sources and targets for `ViewConnections` can be other `ViewConnections`.

An interesting functionality feature of the Graphical Modeler palette is related to its support for Profiles: its dynamic listing of the available Profiles and Stereotypes. The palette lists all available Profiles as palette drawers, and all Stereotypes as palette tools contained within the corresponding palette drawer. `UMLModelerPaletteFactory` (explained further down this subsection) adds the palette as an observer of the model (i.e., the `ModelContents`), and the palette adds itself as an observer of all `Profiles`; this allows the palette to detect the addition and removal of any `Profiles` or `Stereotypes`, and to update its listing of Profiles and Stereotypes accordingly.

This package also defines the factory class `UMLModelerPaletteFactory`, which provides the static method `CreatePalette` that receives a `Diagram` and a `ModelContents` and returns an instance of `UMLModelerPalette`. This palette instance is then used by an

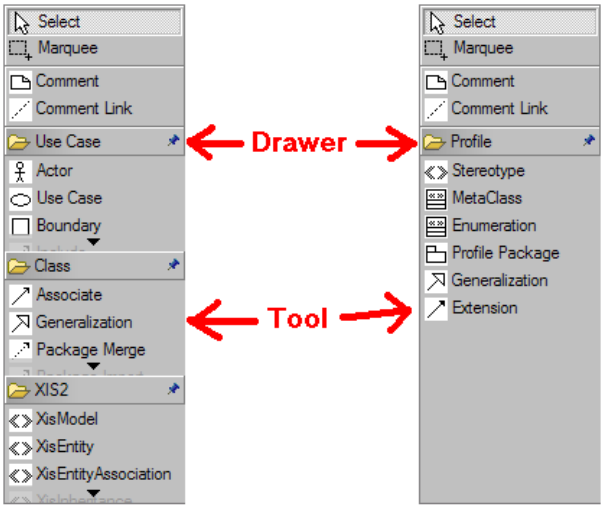

Figure 4.20: Examples of the palette, with palette drawers and palette tools.

instance of the Graphical Modeler (i.e., an editor of a diagram). This factory also adds the palette as an observer of the model so that, if the `Diagram`'s owner is changed from an element contained in a `RootNode` to an element contained in a `Profile` (or vice-versa), the palette will automatically update the available elements to reflect the new owner. Figure 4.20 shows two screenshots of the palette: the left screenshot shows the palette of a diagram owned by an element that is contained in a `RootNode`, and the right screenshot shown the palette of a diagram owned by an element contained in a `Profile`.

Actions

The **Actions package** is similar in nature to the actions defined in the Outline Page, as it defines a set of possible operations presented to the user when a right-mouse-button click occurs over the Graphical Modeler. Figure 4.21 presents an example of the context menu that is presented when the user right-clicks over the representation of an `Actor`.

Similarly to Outline Page, this package defines the class `DiagramAction`, which inherits from the abstract class `SelectionProviderAction` provided by Eclipse.NET, and represents each of the available options in the diagram's context menu. In order to allow the grouping of actions into logical groups (e.g., a group of actions related to a `ViewNode`, or a group of actions related to a `StereotypeApplication`), this package also defines the class `DiagramActionGroup`, which inherits from the abstract class `ActionGroup` provided by Eclipse.NET, and is nothing more than an aggregation of `DiagramActions`. This package also defines the class `DiagramContextMenuActionGroup`, which inherits from `DiagramActionGroup`, and is supposed to be used as a Singleton [Gamma 95]; this class

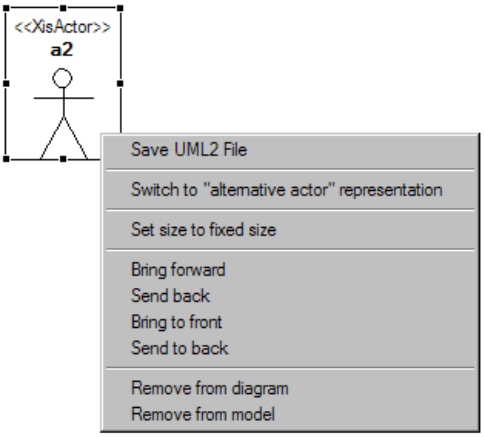

Figure 4.21: Screenshot of Graphical Modeler's context menu.

aggregates all `DiagramActionGroup` classes defined in UMLModeler, and *forwards* all calls to its `FillContextMenu` method to each of those classes.

Finally, GEF defines an abstract class, `ContextMenuProvider`, which must be used by plugins in order to provide context menus in their graphical editors; this package defines the class `DiagramContextMenuProvider`, which inherits from `ContextMenuProvider` and merely forwards "context menu build" requests (made through the `BuildContextMenu` method) to the `DiagramContextMenuActionGroup` class.

Figure 4.22 presents the main classes defined by the Actions package.

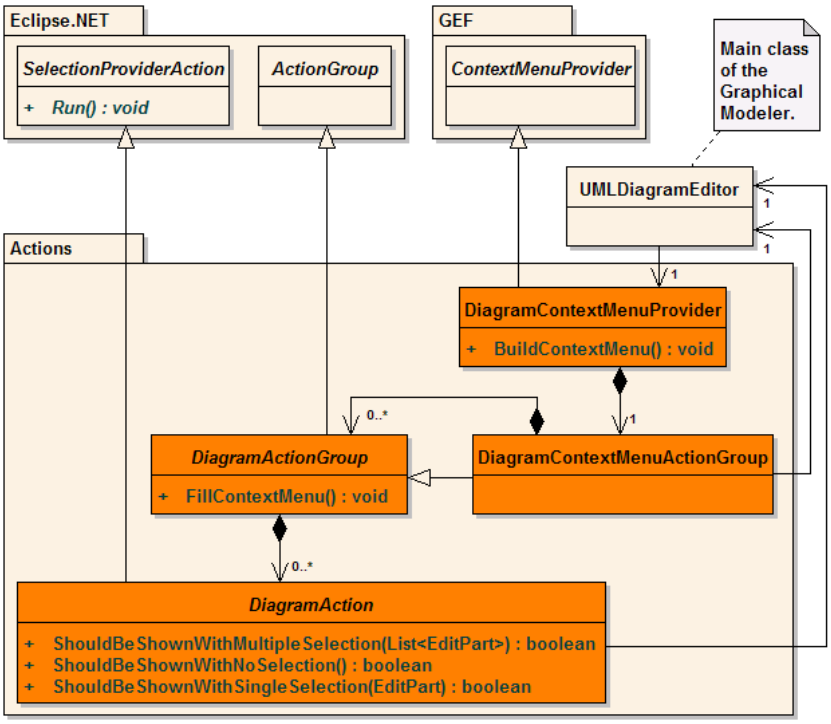

Figure 4.22: Overview of the classes of the Actions package.

It is important to highlight three methods defined by `DiagramAction` (which are named `ShouldBeShownWithoutSelection`, `ShouldBeShownWithSingleSelection`, and `ShouldBeShownWithMultipleSelection`), that indicate whether the action should be shown in the context menu, according to the elements selected in the Graphical Modeler when the context menu was invoked; this is similar to what happens with the class `Action` defined in the Outline Page, which defines the method `ShouldBeShownInContextMenu` that is used for the same purpose.

The justification for the existence of two separate sets of actions (the actions in the Outline Page and the actions in the Graphical Modeler) is that these actions are used in *different contexts*: while the context of Outline Page actions contains *only* a certain UML element, the context of Graphical Modeler actions includes a diagram being edited, and possibly the visual representation of an UML element (depending on whether the user right-clicks over an UML element or over a "blank" section of the diagram). Nevertheless, some of the actions from both sets have the same purposes (e.g., the "Save UML model" actions); in these cases, the corresponding functionality is refactored into external classes that are used by both sets of actions.

Requests

The **Requests package** contains all the Request classes defined by UMLModeler. This package does not define any base classes for requests, as GEF already provides a wide variety of Request classes that allow all the typical operations of a generic graphical modeler.

Nevertheless, this package contains one Request class, `PlaceNodeRequest`, which inherits from the class `CreateRequest` (provided by GEF). The semantics associated with `PlaceNodeRequest` specifies that the user intends to insert a visual representation of an already-existing UML element into a diagram; the user typically does this operation by *dragging the element from the model outline to the diagram editor*. In UMLModeler's case, an instance of `PlaceNodeRequest` is created when the user drags an element from the Outline Page to the Graphical Modeler.

Commands

The **Commands package** contains all the commands defined in UMLModeler, which can be used for any number of purposes, such as editing the current diagram, applying a stereotype to an element, or editing the properties of a certain UML element (e.g., setting a Class as abstract). Commands are obtained by UMLModeler through its interpretation

of Requests (e.g., the `PlaceExistingEntityCommand` is obtained after UMLModeler interprets the `PlaceNodeRequest`, presented in the "Requests" subsection of this chapter).

Commands implement the Command design pattern [Gamma 95], thus allowing the user to undo (and/or redo) previously invoked commands. Figure 4.23 presents an overview of the structural architecture of the Commands package.

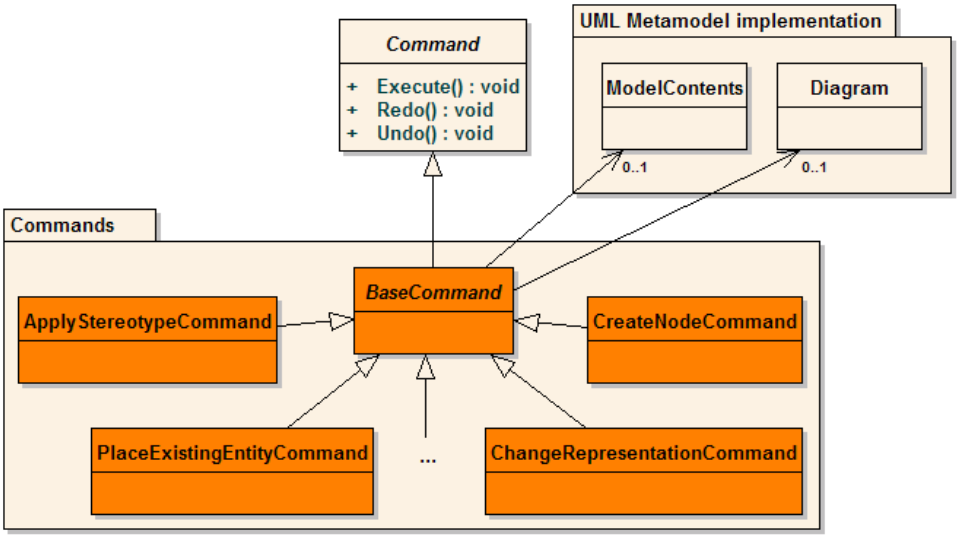

Figure 4.23: Overview of the classes of the Commands package.

All commands inherit from the `BaseCommand` abstract class, which inherits from the abstract class `Command` provided by GEF. `BaseCommand` is used for refactoring common functionality in UMLModeler's commands, and currently only provides references to the model being edited (i.e., the `ModelContents` currently opened in UMLModeler) and to the diagram from which the command was invoked; concrete command classes are responsible for supplying these values to `BaseCommand`'s constructor.

Commands are always validated by UMLModeler before they are executed. When the command depends only on the user making a single mouse-click over the diagram or an element (which is the case with the `CreateNodeCommand` and the `ApplyStereotypeCommand`), this validation occurs while the user is moving the mouse, and the validation's result is presented as visual feedback. A good example of this kind of validation is the application of a stereotype to an element (i.e., the `ApplyStereotypeCommand`): when the user moves the mouse (with the stereotype selected in the Palette) over an UML element, such as a Class, UMLModeler will provide visual feedback indicating whether the stereotype can be applied to the Class (if the user can not apply the stereotype to the Class, a "forbidden" mouse cursor appears).

This package currently contains a wide range of functionalities, although the more complex commands still require extensive testing by ProjectIT-Studio's users in order to detect bugs that may occur in circumstances not expected by UMLModeler.

4.2.3 Property Forms

Double-clicking on an element (either in the Outline Page or in the Graphical Modeler) is interpreted by UMLModeler as a wish to open a **Property Form** containing all the details about the selected element. Figure 4.24 shows a screenshot of such a form (in this case, it presents the details of an UML Class).

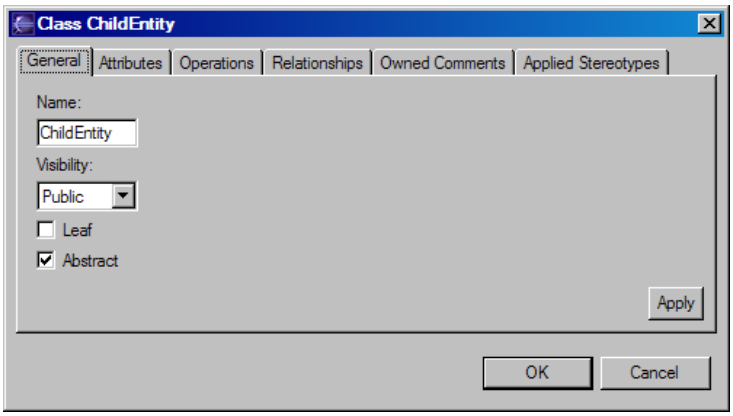

Figure 4.24: Screenshot of a Property Form for an UML Class.

Nevertheless, there is a great number of elements defined in the UML Superstructure, many of which share characteristics because of the inheritance relationships between them. Thus, it was necessary to create a mechanism that allows the *refactoring* of the visual elements (e.g., labels, text boxes, check boxes) necessary to edit the information defined by each of the UML Superstructure's elements. Figure 4.25 presents an overview of the classes involved in this mechanism.

The `PropertyFormFrontController` class is responsible for receiving the object to be edited (usually an UML element) and opening the corresponding Property Form that will edit that object; as its name suggests, this class can be considered as an implementation of the **Front Controller design pattern** [Fowler 03].

`TabbedForm` is the base class for all Property Form classes, and it provides the functionality required for creating tabs that will be contained in the form; these tabs are added to the form in the `FillForm` method. Each tab inherits from `BaseTab`, which provides all functionality necessary to define a tab that belongs to a `TabbedForm`. Nevertheless, a tab can instead inherit from `BaseGeneralTab` (which in turn inherits from `BaseTab`); this

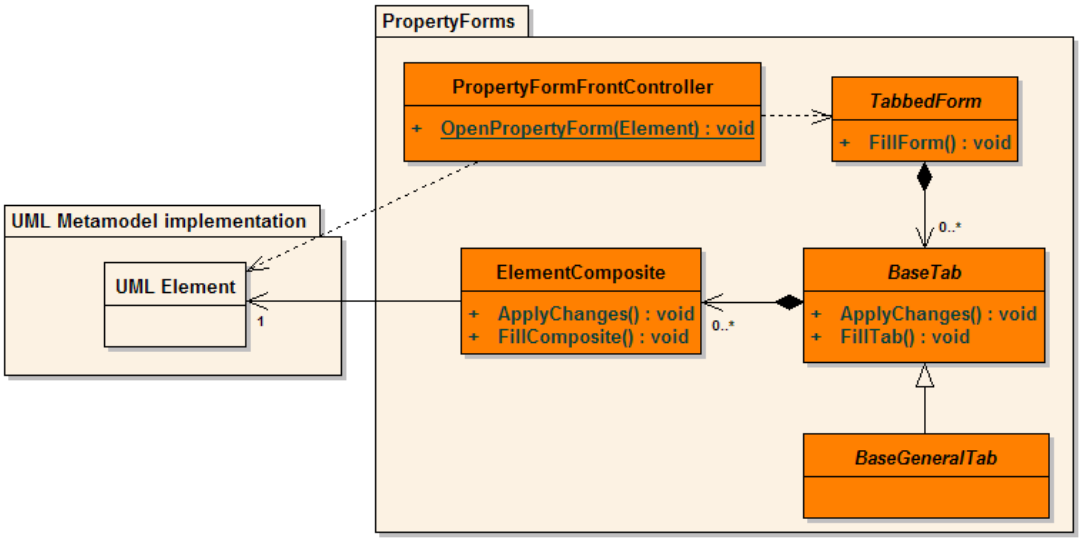

Figure 4.25: The classes that support the Property Forms mechanism.

class is typically used to define the "General" tab for each UML element, and it includes an "Apply" button in the lower right corner of the tab (as shown in Figure 4.24).

Additionally, each `BaseTab` can contain any number of `ElementComposite`s, which are added to the tab by the `FillTab` method. An `ElementComposite` is nothing more than a group of visual elements (such as labels and text boxes) that allow the visualization and editing of the corresponding UML element's attributes (e.g., "NamedElement" defines the attributes "name" and "visibility", so there is a class, called `NamedElementComposite`, that inherits from `ElementComposite` and consists of a group of visual elements that reflect a NamedElement's name and visibility).

Figure 4.26 depicts the screenshot that was first presented in Figure 4.24, now highlighting these concepts (note that the names shown in the figure are not the names of the concrete classes, but of the base classes presented in this subsection).

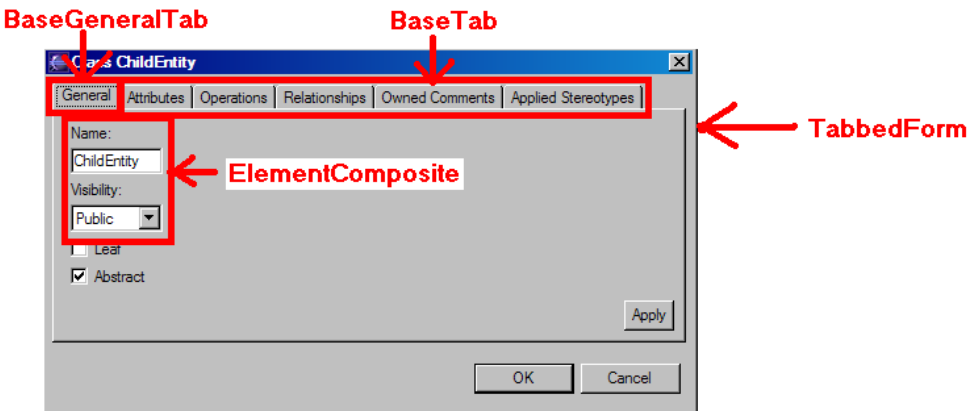

Figure 4.26: The components of a Property Form.

Tabs other than the "General" tab usually contain lists of child elements (e.g., the Property Form for a Class has tabs that show the lists of Attributes and Operations of that Class). Moreover, these tabs also provide a set of possible actions that can be performed. An example of this is presented in Figure 4.27, that presents a tab with a list of the Attributes of the current Class; this tab provides the actions "New", "Edit", and "Delete" (which respectively create a new Attribute, edit the Attribute currently selected in the list, or delete it), and the possibility of changing the order of the Attributes, by using the arrow buttons located at the right side of the list.

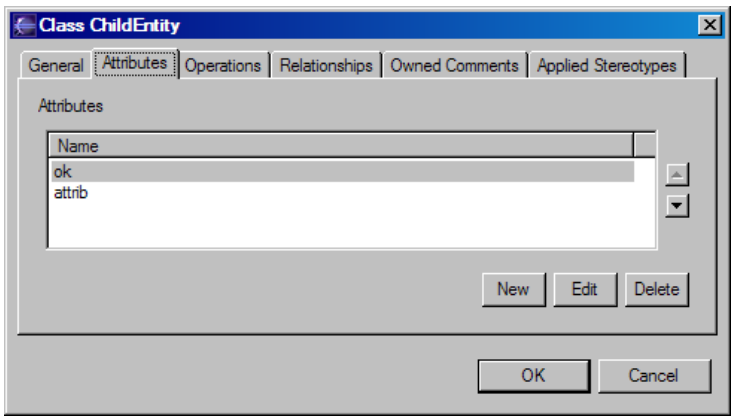

Figure 4.27: A tab, with a list of Attributes, in a Property Form of an UML Class.

Additionally, UMLModeler allows the *quick renaming* of the elements in a list, without requiring that the user open the corresponding Property Form and change the name in that Form. To do a quick renaming of an element in a list, the user needs only to *double-click the mouse over the name of the element* and a text box will appear over the name, allowing the user to change the element's name.

UMLModeler also provides a special tab, called `AppliedStereotypesTab`, that shows a list of all the `StereotypeApplications` corresponding to the element being edited by the Property Form (i.e., all the stereotypes applied to that element). This tab allows the following operations: (1) create another application of a Stereotype to the current element (i.e., create another `StereotypeApplication`); (2) change the selected `StereotypeApplication` to a different Stereotype; (3) remove the currently selected `StereotypeApplication`; and (4) edit the tagged values defined by the corresponding Stereotype (this action can also be quickly accessed by *double-clicking* the mouse over the corresponding `StereotypeApplication`). Figure 4.28 shows an example of `AppliedStereotypesTab` (at the top of the figure) and of the Property Form that corresponds to a `StereotypeApplication` (at the bottom of the figure).

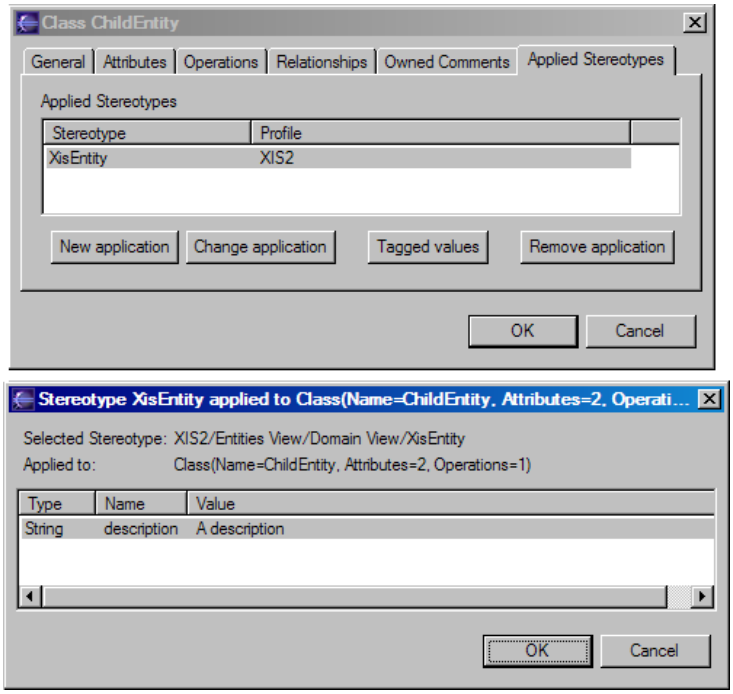

Figure 4.28: The "Applied Stereotypes" tab (top) and a `StereotypeApplication`'s Property Form (bottom).

4.2.4 Preferences

UMLModeler uses the **Preferences** mechanism offered by Eclipse.NET, which requires that a plugin developer perform two actions: (1) define a class that implements the `UI.IWorkbenchPreferencePage` interface provided by Eclipse.NET; and (2) register that class as a **preference page** in the plugin's manifest file (by providing an extension to the `UI.PreferencePages` extension point, also provided by Eclipse.NET). After performing these actions, Eclipse.NET's *Preference dialog* will show a new page corresponding to the new preference page. A plugin can have as many preference pages as desired, simply by providing an identical number of classes (and corresponding extensions) that define each of those pages.

UMLModeler currently provides four preference pages of its own: (1) *UMLModeler general preferences*; (2) *diagram preferences*; (3) *toolbox preferences*; and (4) *Outline Page preferences*. These preference pages are organized in an hierarchical structure, with the "UMLModeler general preferences" page acting as the parent of the other three preference pages. Figure 4.29 presents a screenshot of the Preferences dialog, with the preference pages provided by ProjectIT-Studio's plugins highlighted in the left side of the figure; the right side of the figure highlights the options provided by the selected preference page (in this case, "diagram preferences").

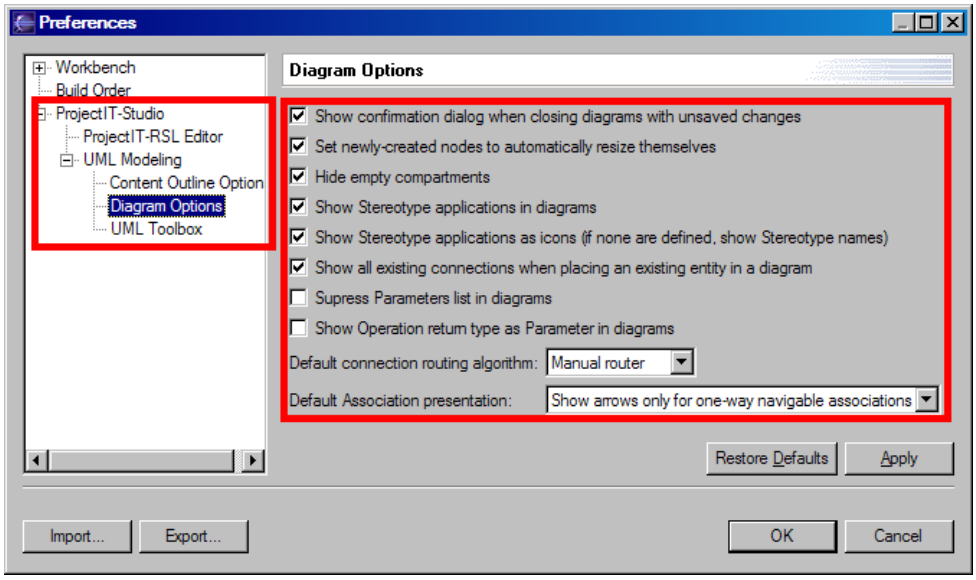

Figure 4.29: Screenshot of the Preferences dialog with ProjectIT-Studio's preference pages.

These preference pages are defined in UMLModeler's **Preferences package**. This package contains an abstract class, `BasePreferencesPage`, which provides a basic mechanism for notifying any interested observers when the value of a preference changes. This package also contains four other classes that inherit from `BasePreferencesPage`: (1) `ToolboxPreferencesPage`, which defines the preferences associated with the Graphical Modeler's Toolbox, consisting mostly of options to display (or remove) the individual components of the Toolbox; (2) `DiagramPreferencesPage`, which defines a wide range of preferences to customize the way a diagram is displayed by the Graphical Modeler; (3) `OutlinePreferencesPage`, which defines the preferences associated with UMLModeler's Outline Page and the way it displays model information; and (4) `GeneralPreferencesPage`, which defines preferences that are not particular to a specific component of UMLModeler, since they apply to the entire UMLModeler plugin. Figure 4.30 presents a structural overview of this package.

It should be noted that this package's external dependencies are used only to obtain some specific values (e.g., the list of available Outline Page sorters), and these dependencies could easily be removed. Also, it is expected that the range of available options will grow considerably as user feedback is received.

4.2.5 Wizards

UMLModeler also uses the **Wizards** mechanism offered by Eclipse.NET. This mechanism requires that the plugin developer perform the following actions: (1) define the

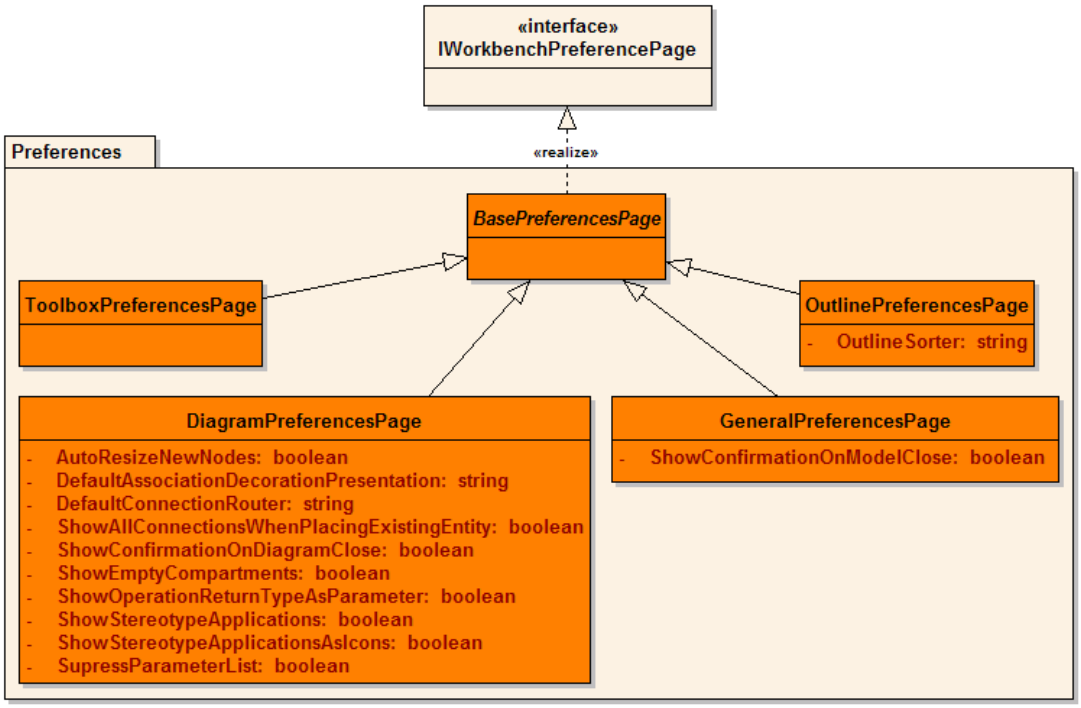

Figure 4.30: The structure of the Preferences package.

classes that represent the various steps (or *pages*) of the wizard, and set them to inherit from the `JFace.Wizard.WizardPage` class provided by Eclipse.NET; (2) define a class that implements the `UI.INewWizard` interface provided by Eclipse.NET; and (3) register that class as a wizard in the plugin's manifest file (by providing an extension to the `UI.NewWizards` extension point, also provided by Eclipse.NET). After performing these actions, Eclipse.NET's *Wizard selection dialog* (shown in Figure 4.31) will show a wizard entry corresponding to the new wizard. A plugin can have as many wizards as desired, simply by providing an identical number of classes (and corresponding extensions) that define each of those wizards.

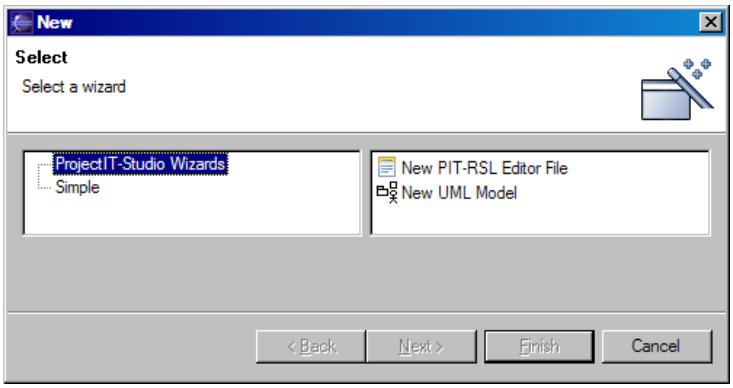

Figure 4.31: Screenshot of the Wizards selection dialog.

The Wizards package defines two abstract classes, `UMLWizard` and `UMLWizardPage`, which form the basis for all UMLModeler wizards; the objective of these two abstract classes is to refactor functionality that will be common to most (if not all) of the UMLModeler wizards. Figure 4.32 presents a structural overview of the Wizards package.

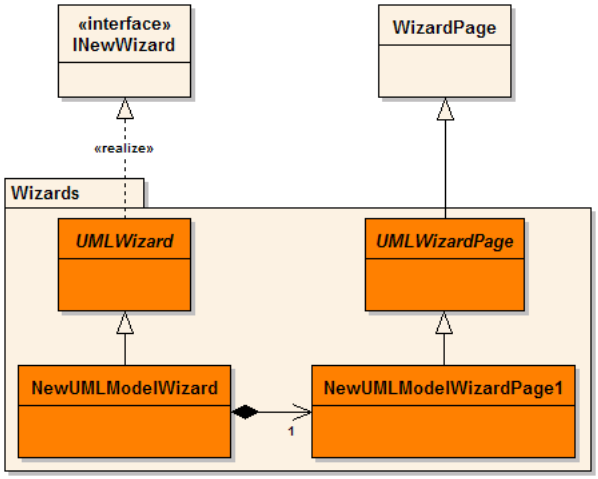

Figure 4.32: The structure of the Wizards package.

This package currently contains a single wizard, `NewUMLModelWizard` (also presented in Figure 4.32), which is used to create a new UML model file. This wizard asks the user where the new model file should be located, and then creates a new model file accordingly; this new model contains a `RootNode`, a `View`, and a `Diagram`. Figure 4.33 presents a screenshot of the `NewUMLModelWizard` and of the outline of the model file that is automatically created by this wizard.

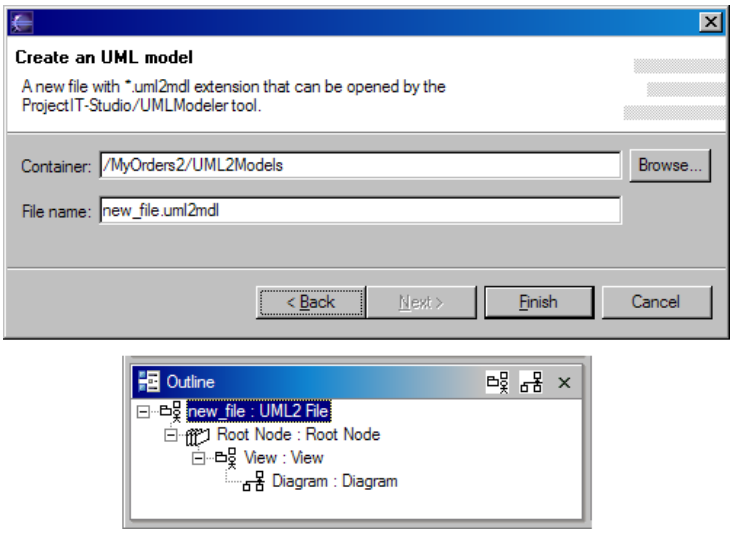

Figure 4.33: Screenshot of the `NewUMLModelWizard` and the outline of the resulting model file.

This package is currently in a very preliminary state, and it is expected that the number of wizards will grow as user feedback is received.

4.3 Integration with ProjectIT-Studio

One of the requirements for UMLModeler was that it should be fully integrated with the other ProjectIT-Studio plugins, to support the ProjectIT approach in a simple and efficient way.

4.3.1 Integration with the Requirements Plugin

The integration between Requirements and UMLModeler was done through an extension point provided by the former. This extension point, called "RequirementsModelProvider", is used by Requirements to provide an UML model obtained from text-based natural language requirements, and it expects extensions to provide a method that receives the following parameters: (1) the UML model; and (2) a file path "hint" indicating the default location where the UML model should be stored.

The UMLModeler plugin provides such an extension. This extension, when invoked, immediately creates an instance of the UML modeling editor, and the UML model is displayed by this editor instance. On the other hand, the file path hint is only used when the Designer tries to store the model (using the typical "Save" or "Save As..." menu options) and the model has not yet been stored (i.e., its `path` attribute is empty); in this case, this hint is used as the initial file path presented by the saving dialog. However, in order to enhance the ProjectIT-Studio user experience, the plugin also establishes a mapping between the model's `sourceIdentifier` attribute and the file path chosen by the Designer; this mapping is checked when a model is provided through the extension, and if a match is found, the model's `path` attribute is updated to reflect the file path previously specified by the Designer.

4.3.2 Integration with the MDDGenerator Plugin

The integration between MDDGenerator and UMLModeler was done in a very similar way to the integration described in the previous subsection. UMLModeler provides an extension point, called "UMLModelProvider", through which it provides an UML model. This extension point expects extensions to provide a method that can receive one of the following sets of parameters: (1) no parameters; (2) an UML model; or (3) an UML model and the Eclipse.NET project (which is a container of files and directories) that contains the file where that UML model is stored (or the file that contains the requirements that

originated the model, if it was provided through the Requirements' extension point). The plugin determines at runtime the signature of the method corresponding to the extension, and invokes the method using the appropriate parameters.

The MDDGenerator plugin provides an extension to the "UMLModelProvider" extension point. This extension provides a method that receives an UML model and the Eclipse.NET project in which it is contained (this project is necessary for MDDGenerator to determine where it will place all its files [Silva 06b]); this method displays a "wizard" which guides the Programmer through the creation of a Generative Process that uses the model supplied by UMLModeler.

UMLModeler presents the various extensions provided to the "UMLModelProvider" extension point as a list of possible actions that can be performed over the UML model. Figure 4.34 presents a screenshot of UMLModeler with such a list (in the context menu); the option to invoke MDDGenerator, through the extension provided by the later, has the text "Create Generative Process" and is highlighted in the figure.

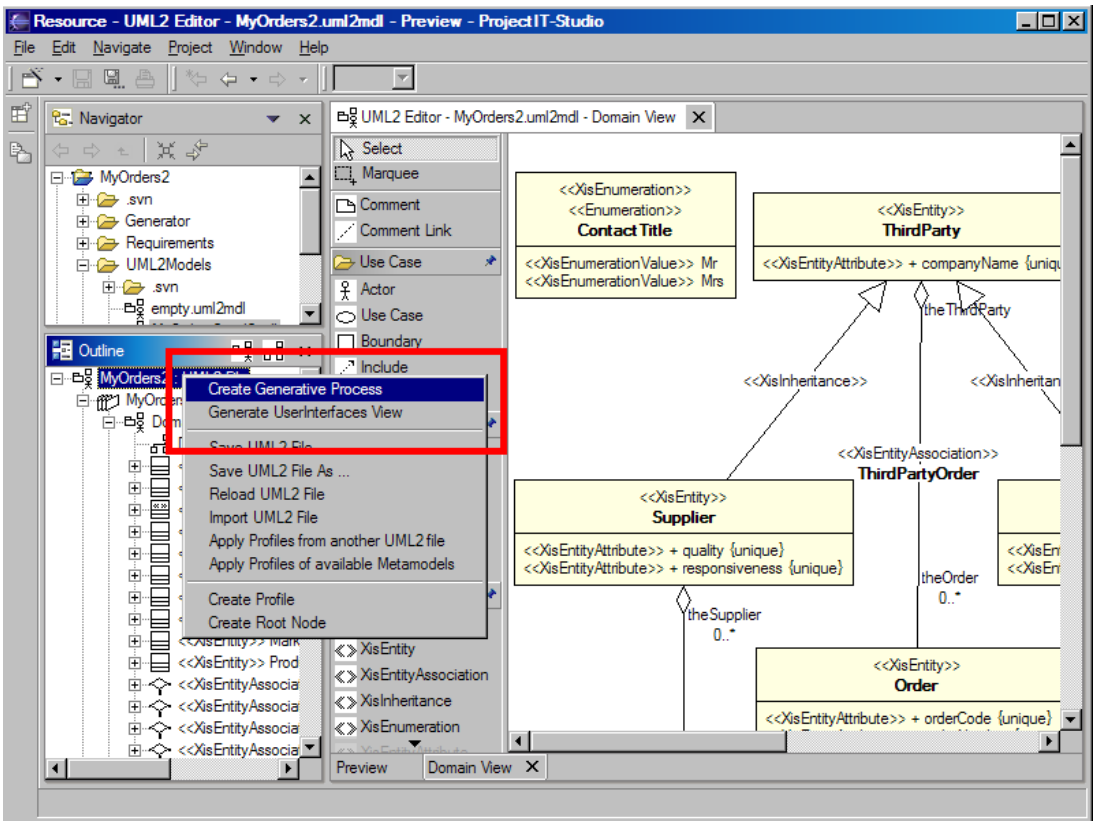

Figure 4.34: List of actions obtained through the "UMLModelProvider" extension point.

A curious remark about this extension point is that it is not only used by MDDGenerator to create Generative Processes, but it can also be used by other plugins to interpret and/or manipulate the UML model. An example of this kind of usage are "Model-to-

Model" transformations, which directly manipulate the UML model itself. This usage is described in detail in the next section.

4.4 Support For UML Model Manipulation By Other Plugins

Model-to-Model transformations are only an example of the *UML model interpretation and manipulation mechanism* provided by UMLModeler, which uses the "UMLModelProvider" extension point presented in the previous section. A detailed explanation of this mechanism requires the presentation of some details about "UMLModelProvider" [2]. An example of an extension to "UMLModelProvider" is presented in Listing 4.1.

Listing 4.1: An extension to "UMLModelProvider" that provides a transformation.

```
<extension point="GSI_INESC.PITStudio.UML_Modeler.UMLModelProvider">
    <run
        type="Xis2.ModelTransformations.GenerateUserInterfacesView"
        method="TransformModel"
        validationMethod="ValidateTransformation"
        caption="Generate UserInterfaces View" />
</extension>
```

Any extension to "UMLModelProvider" can supply four parameters: (1) `type`, which is the fully-qualified name of the class where the transformation methods are defined; (2) `method`, which is the name of the method that defines how the transformation is applied; (3) `validationMethod`, which is the name of the method that determines whether the transformation can be applied; and (4) `caption`, which is a string presented by UMLModeler to the user, that identifies the purpose of the transformation. Of these four parameters, only the `type` and `method` parameters are required, because without them it would not be possible to apply the transformation; the `validationMethod` and the `caption` parameters, although recommended, are not required.

As the previous section mentioned, UMLModeler builds a list of all the extensions to "UMLModelProvider" and presents them to the user as a list of possible actions in the Outline Page's context menu. This work is done through two particular classes defined in the OutlinePage: (1) `ModelTransformationActionGroup`, which inherits from

[2]For text simplicity throughout the rest of this section, the "UMLModelProvider" extension point will be referred to only as "UMLModelProvider", and a model-to-model transformation will be referred to only as "transformation" (unless explicitly stated otherwise). Also, the mechanism is only presented in terms of model-to-model transformations (for text simplicity), although it can also be used for any type of model interpretation and/or manipulation purposes.

`OutlineActionGroup`; and (2) `ModelTransformationInvokerAction`, which inherits from `OutlineAction`. These two classes are illustrated in Figure 4.35.

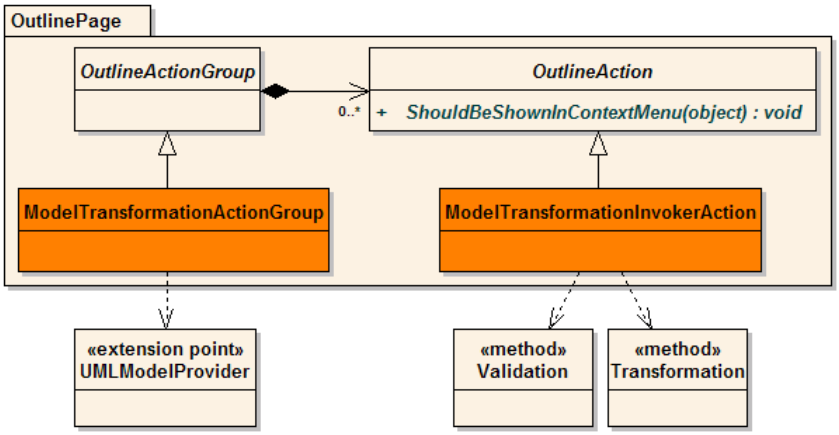

Figure 4.35: The classes involved in UMLModeler's transformation mechanism.

The Outline Page's `ContextActionGroup` (already presented in this chapter) creates an instance of the `ModelTransformationActionGroup` class. When this instance is created, its constructor analyzes the list of extensions to "UMLModelProvider" and, *for each extension that provides all the required information*, creates an instance of the `ModelTransformationInvokerAction` class (supplying the extension's information to the constructor) and stores that instance in its internal list of aggregated actions.

Each `ModelTransformationInvokerAction` is an action that represents a transformation. This class stores the information that is necessary to invoke (through reflection) the validation and transformation methods of the corresponding extension, by defining the following C# *delegates* (which can be considered as "pointers to methods"): (1) `TransformInvoker`, which allows the invocation of the transformation method; and (2) `TransformValidationInvoker`, which allows the invocation of the validation method.

The `ModelTransformationInvokerAction` constructor receives all the strings corresponding to the extension's parameters (i.e., `type`, `method`, `validationMethod`, and `caption`), validates those strings by verifying (through reflection) if the methods indicated by those strings exist, and creates instances of the appropriate delegates (i.e., it creates an instance of the `TransformInvoker` delegate and, if the validation method was specified in the extension, an instance of the `TransformValidationInvoker` delegate).

Listing 4.2 shows an interesting part of `ModelTransformationInvokerAction`'s source code, which shows how these two delegates are used by UMLModeler to determine whether the transformation can be applied to the model (by using the validator method), and to perform the transformation (by using the transformation method).

Listing 4.2: Part of the `ModelTransformationInvokerAction`'s source code.

```
public class ModelTransformationInvokerAction : OutlineAction {
    private delegate bool TransformValidationInvoker(ModelContents mc);
    private delegate void TransformInvoker(ModelContents mc);

    private ModelContents model;
    private TransformValidationInvoker theValidatingDelegate;
    private TransformInvoker theTransformationDelegate;
    ...
    public override bool ShouldBeShownInContextMenu(object selection) {
        if(theTransformationDelegate == null) {
            return false;
        }
        if(theValidatingDelegate != null) {
            return theValidatingDelegate(model);
        }
        return true;
    }

    public override void Run() {
        if(theTransformationDelegate != null) {
            theTransformationDelegate(model);
        }
    }
}
```

It is important to note the bodies of the `ShouldBeShownInContextMenu` and `Run` methods specified in Listing 4.2. The `Run` method is invoked when the user clicks over the corresponding context menu item, and it only invokes the transformation method. However, the `ShouldBeShownInContextMenu` method is responsible for determining *whether the menu item should be visible*, and performs the following sequence of operations: (1) check if the transformation method exists (otherwise, some error has occurred and the user should not be able to perform the transformation); (2) if there is a validation method, invoke it to determine if the transformation should be available; and (3) if neither of the two previous checks failed, return `true` to indicate that the menu item should be visible. Note that the `Run` method does not need to perform any validations because it can only be invoked if the `ShouldBeShownInContextMenu` method returns true (otherwise, the user would not even see the menu item).

Listings 4.3 and 4.4 present very basic examples of a validator method and a transformation method, respectively. These methods were previously declared in Listing 4.1, and are used for the transformation defined by the "smart" design approach of the XIS2 Profile [Silva 07]. The validator method only checks if the XIS2 `Profile` is applied to

the given model, and if there are any `RootNodes` containing model elements to be transformed; on the other hand, the transformation method itself invokes a series of utility methods that process and modify the model (the specifics of the "smart" approach and its transformation are outside the scope of this thesis; for further information, please consult [Silva 07]).

Listing 4.3: Example of a validation method.

```
namespace Xis2.ModelTransformations {
    public class GenerateUserInterfacesView {
        ...
        public static bool ValidateTransformation(ModelContents model) {
            if(!umlModel.Profiles.Exists(
                delegate(Profile profile) {
                    return profile.Name == XIS2_PROFILE_NAME; })) {
                // XIS2 Profile was not applied to the model.
                return false;
            }
            // Also do some useful verifications.
            List<RootNode> rootNodes = umlModel.RootNodes;
            if(rootNodes == null || rootNodes.Count == 0) {
                return false;
            }
            return true;
        }
    }
}
```

Listing 4.4: Example of a transformation method.

```
namespace Xis2.ModelTransformations {
    public class GenerateUserInterfacesView {
        ...
        public static void TransformModel(ModelContents umlModel) {
            // Perform validation (to ensure good model conversion...)
            if(!ValidateTransformation(umlModel)) {
                return;
            }

            XisModel xisModel = new XisModel();

            GetXisModel(umlModel, xisModel);
            GetXisEntitiesView(umlModel, xisModel);
            GetXisUseCasesView(umlModel, xisModel);

            GenerateUserInterfaces(xisModel);
```

```
17
18                TransformUserInterfacesViewToUML2(xisModel, umlModel);
19            }
20        }
21    }
```

Currently, UMLModeler does not provides a mechanism to define *how* a transformation should be applied. Although UMLModeler supports the application of these validations and transformations to a model, these validations and transformations are specified only through source code, and transformations with a high degree of complexity can easily reveal themselves as very complex to specify through source code alone (with no model transformation primitives to assist in this specification). Another shortcoming of UMLModeler is that it does not provide any mechanism to handle errors that occur during the transformation. Although the validation method is supposed to ensure that the transformation can occur without errors, it would be very naive to assume that such validations support *all* possible conditions that might originate transformation errors. These are some current restrictions that will be addressed in the near future.

4.5 Other Functionalities

Besides the architectural aspects presented in this chapter, UMLModeler also provides a range of additional functionalities (some of which are based on the UML modeling tool features mentioned in "State of the Art", Chapter 2) that should considerably improve its quality and usefulness.

Some examples of these functionalities are: (1) *use of the .NET Framework's localization mechanism*, which allows the internationalization of UMLModeler (i.e., changing the language in which UMLModeler presents its user-interface) by simply placing a "satellite assembly" (containing resource strings in the appropriate language) [.NET a] in the directory where UMLModeler is installed; and (2) *exporting a diagram to an image file*, which allows the Designer to store the visual layout of a diagram in an image file (due to limitations in SWT's implementation, a diagram can currently be exported only to JPEG, BMP or ICO formats).

Since these functionalities are based only in particular technological features of the various frameworks used in this work and not on a particular architectural design feature of UMLModeler, these functionalities are not explained in detail in this chapter. However, further information about them can be found in "Appendix B – ProjectIT-Studio Designer's Manual" and "Appendix C – ProjectIT-Studio Programmer's Manual".

Chapter 5

Supporting the XIS2 UML Profile

The main goal of UMLModeler is to support the modeling tasks of the ProjectIT approach. Since the ProjectIT approach is language-independent (i.e., it does not depend on a particular language being used), UMLModeler could not be developed having a specific language in mind, but rather a language definition formalism such as the UML Profile mechanism.

The "eXtreme modeling Interactive Systems" UML profile (XIS2) [Silva 07] was used to evaluate the Profile definition functionalities of UMLModeler, due to the following reasons: (1) XIS2 covers most aspects of the software system development life-cycle, from domain specification to user-interface design; (2) the language defined by XIS2 consists exclusively of UML stereotypes; (3) XIS2 proposes a relatively complex "model-to-model" transformation to accelerate the modeling of a software system; and (4) XIS2 uses some sketching techniques, based on the layout of UML diagrams, to specify the design of the user-interface.

This chapter presents the process of defining the XIS2 profile in UMLModeler. First, a brief overview of the XIS2 Profile is given. Afterward, the definition of the XIS2 profile and the "model-to-model" transformation in UMLModeler is presented. Finally, this chapter ends with some examples of the application of the XIS2 language (in this case, the model of the "MyOrders2" software system, which is explained in detail in "Appendix A – The "MyOrders 2" Case Study").

5.1 Brief Overview of the XIS2 UML Profile

XIS2 is an UML Profile that is oriented towards the development of interactive software systems [Silva 07], and its main goal is to allow the modeling of the various aspects of an interactive software system, in a way that should be as simple and efficient as possible.

To achieve this goal, XIS2 design follows some key ideas [Silva 07]: (1) *modularization*, by allowing the definition of "business entities" with any level of granularity; (2) *separation of concerns*, through the definition of multiple views that handle different aspects and are relatively independent among themselves; (3) *use-case-driven approach*, through the identification of actors and use cases, to manage the main functionalities of the system; and (4) *model transformations*, by defining a series of possible approaches based on different kinds of model transformations (e.g., "XIS2 model"-to-"source code" or "XIS2 model"-to-"XIS2 model"). Although XIS2 currently defines two different approaches (the "dummy" approach and the "smart" approach), ProjectIT-Studio provides complete support only for the "dummy" approach, and support for the "smart" approach is currently under development.

5.1.1 XIS2 Multi-Views

XIS2 addresses the development of a system as the modeling of that system according to multiple views, which are illustrated in Figure 5.1.

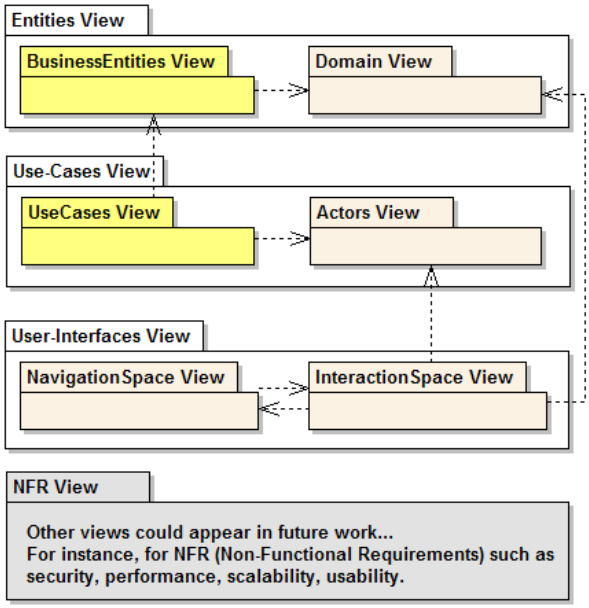

Figure 5.1: The multi-view organization of XIS2 (extracted from [Silva 07]).

The **Entities View** specifies the domain of the system in various levels of granularity. This view is composed by the following views: (1) the **Domain View**, which specifies the entities relevant to the system's problem-domain in a traditional way (by using classes, attributes, and relationships such as associations, aggregations, and inheritance); and (2) the **BusinessEntities View**, which specifies the system's business entities (i.e., the "high-level" entities that are manipulated by the end-user) as aggregations of domain

entities or even other business entities. Note that *the BusinessEntities View allows the specification of entities of any level of granularity*, because business entities can be of a coarser or finer granularity simply by aggregating (in a composite manner) other business entities or domain entities.

The **Use-Cases View** specifies actors and use cases, and establishes the corresponding permissions. This view is composed by the following views: (1) the **Actors View**, which specifies the system's actors (i.e., the roles that the end-users can assume) that can perform operations, and inheritance relationships between those actors; and (2) the **UseCases View**, which specifies the relationships between actors and the operations those actors are allowed to perform on business entities.

The **User-Interfaces View** specifies the aspects related to the user interface that the system should present to the end-user, and is based on the application of typical interaction patterns [Silva 07]. This view is composed by the following views: (1) the **InteractionSpace View**, which employs sketching techniques to visually define the user-interface interaction elements that are contained in each interaction space (i.e., an abstract "screen" that receives and presents information to the end-users during their interaction with the system), and to specify access control between actors and user-interface elements; and (2) the **NavigationSpace View**, which defines the navigation flow between any of the interaction spaces, in a way similar to a state machine.

A more detailed description of the XIS2 profile is provided by [Silva 07], and its reading is recommended. Some additional information about the XIS2 profile can also be found in [Silva 06b].

5.1.2 Design Approaches

When using the **"dummy" approach** (which is shown in the left section of Figure 5.2), the Designer should produce: (1) the Domain View, (2) the Actors View, (3) the NavigationSpace View; and (4) the InteractionSpace View. Of course, the other views (which are optional) can also be produced, as they could be useful from the documentation perspective; however, these views are useless for the XIS2 model-to-code transformations. It should be noted that this approach is time-consuming (because the NavigationSpace View and the InteractionSpace View can take a long time to model) but can be necessary if the XIS2 model-to-model transformations are not available.

When using the **"smart" approach** (which is shown in the right section of Figure 5.2), the Designer only has to produce: (1) the Domain View, (2) the BusinessEntities View, (3) the Actors View; and (4) the UseCases View. The objective of this approach (relatively to the "dummy" approach) is to accelerate the modeling of the system, by

using "model-to-model" transformations to automatically generate the time-consuming user-interface models. In addition, the generated user-interface models can be customized or changed in order to support specific requirements, i.e., requirements not captured that can be supported by the involved "model-to-model" transformation patterns.

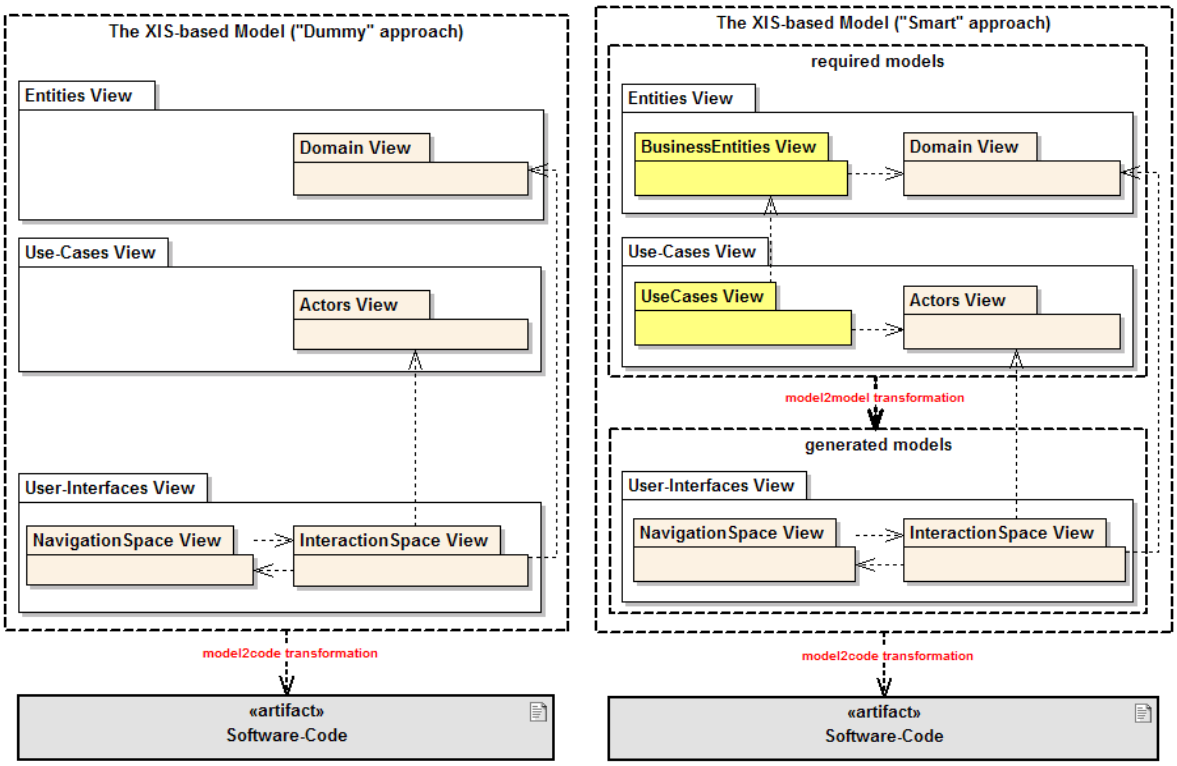

Figure 5.2: The "dummy" and "smart" design approaches defined by XIS2 (extracted from [Silva 07]).

5.2 Defining the XIS2 Profile

The process of defining a profile in UMLModeler involves only the creation of a `Profile` within a model, and the creation of `ProfilePackages` (to allow the segmentation of the contents of a profile into smaller packages) and `Stereotypes` (to allow the extension of the UML language). In order to improve the reusability of `Profiles`, it is usually a good idea to define each `Profile` in a separate "blank" UML model file, and at a later time apply the `Profile` to another UML model file.

After creating a new UML model file by using the `NewUMLModelWizard` provided by UMLModeler, the user can create a new `Profile` by: (1) right-clicking on the UML model (the top-level element) in the Outline Page; (2) selecting the "Create Profile" action from the context menu; and (3) entering the new profile's name (in this case, "XIS2").

After creating the `Profile`, the user can start adding elements (such as `Diagrams`, `ProfilePackages`, or `Stereotypes`) to it. Since XIS2 is organized in multiple views, the user should create three `ProfilePackages` within the `Profile`, corresponding to the Entities View, the Use-Cases View, and the UserInterfaces View. Afterward, the user should also create additional `ProfilePackages`, corresponding to the views that are contained within the previous three views. After all the needed `ProfilePackages` are created, the user can then create `Diagrams` (typically within the `ProfilePackages` that will contain `Stereotypes`). Figure 5.3 shows the Content Outline of the UML model file after these `ProfilePackages` and `Diagrams` are created. Note that a `ProfilePackage` is only a container of `Stereotypes` and other `ProfilePackages`, and is not associated with any particular semantics.

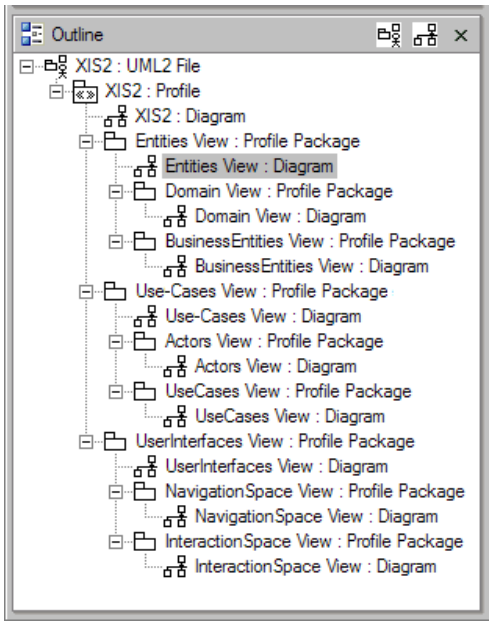

Figure 5.3: The tree structure of the `ProfilePackages` corresponding to XIS2 views.

Finally, the user can start adding `Stereotypes` to the XIS2 `Profile`. To do this, the user needs to do the following for each of the `ProfilePackages` that will contain `Stereotypes`: (1) open the corresponding `Diagram`; (2) create the `Stereotypes`, by selecting the "Stereotype" tool in the Palette and placing it in the Graphical Modeler; (3) if the `Stereotypes` provide any tag definitions (represented in UML as `Attributes` contained by the `Stereotype`), then open the `Stereotype`'s Property Form (by double-clicking on the `Stereotype` in the Graphical Modeler or in the Content Outline) and add the corresponding `Attributes` (in the "Attributes" tab); and (4) for each `Stereotype`, specify the `Stereotypes` that it should extend. This last step involves the following steps: (1) selecting the "MetaClass" tool; (2) clicking on the Graphical Modeler to place the `MetaClass`; (3) choosing the `MetaClass` from the options that are available in the

"Choose Metaclass type" window that appears, and pressing the "OK" button to confirm; and (4) using the "Extension" tool to establish an *extension relationship* between the `Stereotype` and the `MetaClass`. Figure 5.4 shows another screenshot of UMLModeler, illustrating the `Stereotypes` of the XIS2 Domain View, their tag definitions, and the extended metaclasses.

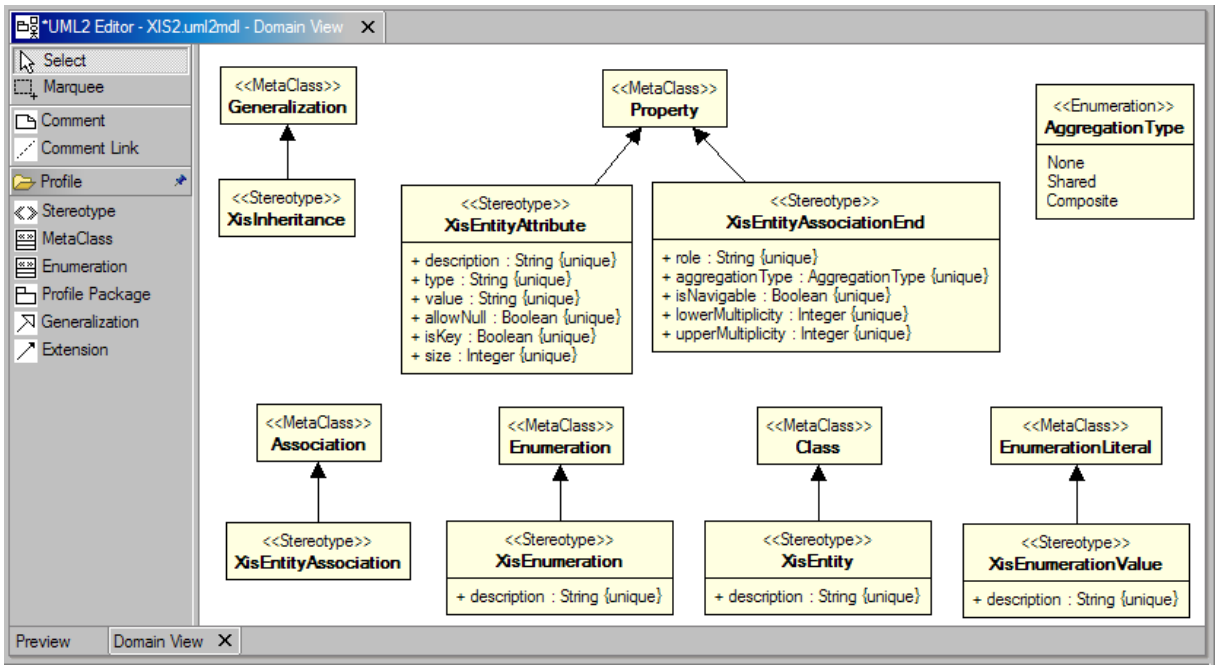

Figure 5.4: The `Stereotypes` of the XIS2 Domain View defined in UMLModeler.

The user can also specify a set of alternative images for each `Stereotype` by performing the following steps: (1) open the `Stereotype`'s Property Form; (2) select the "Icons" tab; and (3) add images to the list by pressing the "New" button and entering the path to the image (images can be removed from the list by selecting them and pressing the "Delete" button). The tab also displays a preview of each icon, for user convenience.

After the user defines the `Profile` (and its `Stereotypes`) and saves the UML model file, the `Profile` is available to be applied to other UML model files.

5.3 Using the XIS2 Profile

After defining the XIS2 profile and saving it into a file, the user can then apply the profile to a model. To do this, the user needs to perform the following steps: (1) open (or create) a model; (2) in the Content Outline, right-click on the tree item corresponding to the UML model, and select the "Apply Profiles from another UML2 file" action; (3) enter the path to the file containing the profile previously defined; (4) in the "Choose Profiles to

apply" window that appears, place the profiles to apply to the model in right box (in this case, the selected file contains only the XIS2 profile, and so this profile is automatically added to the right box); and (5) press the "OK" button. Figure 5.5 presents a screenshot of the "Choose Profiles to apply" window, which allows the user to decide which of the available profiles, obtained from the selected file, will be applied to the current model.

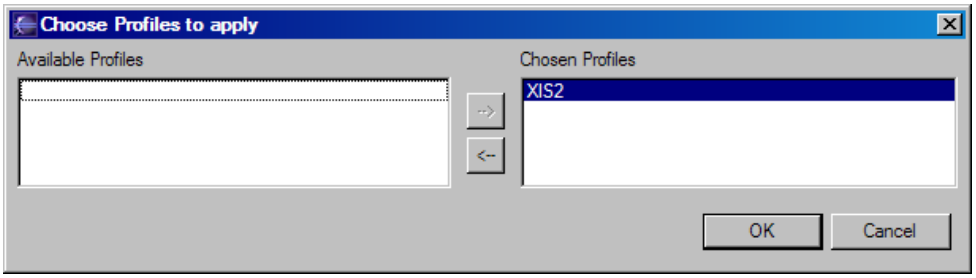

Figure 5.5: Screenshot of the "Choose Profiles to apply" window.

After applying the profile to the model, the user can then apply stereotypes to UML elements. To apply a stereotype to an UML element in a diagram, the user needs only to: (1) select the appropriate stereotype from the Palette; and (2) left-click on the UML element to which the stereotype should be applied. Alternatively, the user can apply a stereotype through the "Applied Stereotypes" tab in the element's Property Form. Figure 5.6 presents a screenshot of the "MyOrders2" Actors View, highlighting: (1) the XIS2 profile applied to the model; (2) the *XisActor* stereotype available in the Palette; and (3) an UML `Actor` to which the *XisActor* stereotype has been applied.

It is important to note that UMLModeler only allows the application of a stereotype to an UML element if that stereotype extends that element's metaclass. Otherwise, the stereotype application will not be allowed, and the user will be notified of this fact through visual feedback (a "forbidden" sign when moving the mouse over the UML element, after the stereotype has been selected in the palette).

Figure 5.7 presents another screenshot, now of the "MyOrders2" Domain View, as this diagram is relatively more complex than the one of the Actors View and provides a better idea of the typical complexity of the models that UMLModeler can handle.

Finally, the "model-to-model" transformation defined by the XIS2 "smart" approach can be applied to the model simply by selecting the "Generate UserInterfaces View" action from the model's context menu. This transformation will generate the UserInterfaces View [Silva 07] according to the defined XIS2 stereotypes and the UML elements to which these stereotypes have been applied.

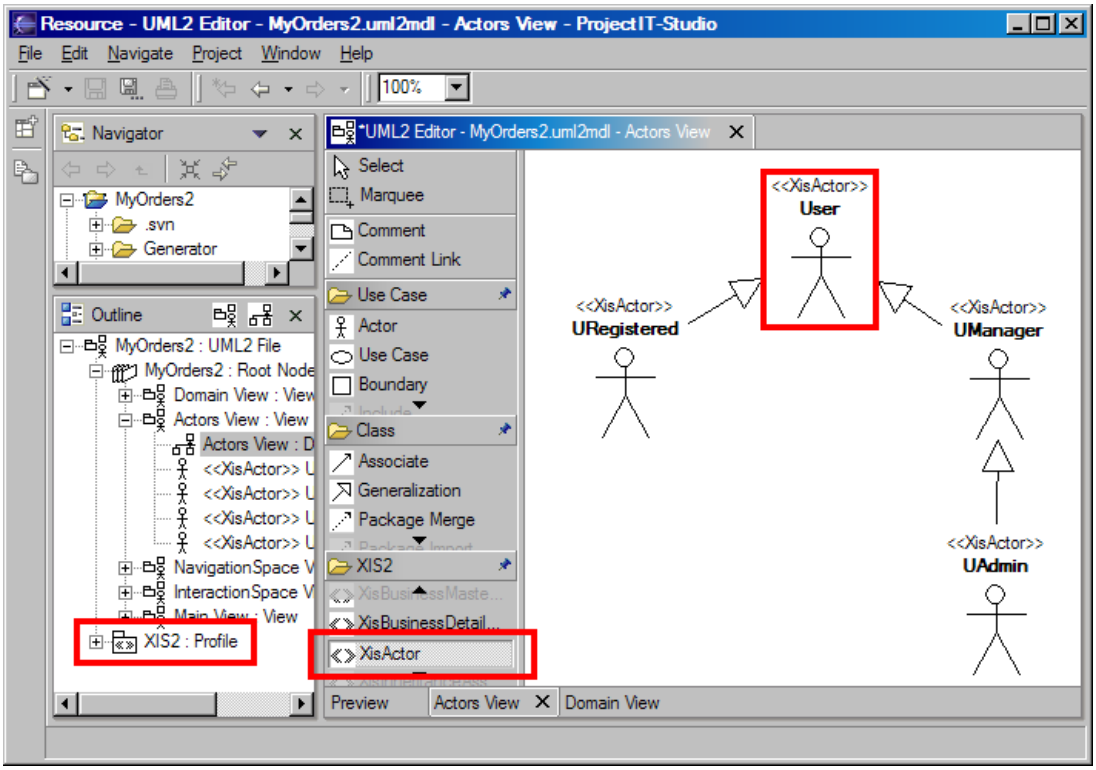

Figure 5.6: Screenshot of the "MyOrders2" Actors View with the XIS2 profile.

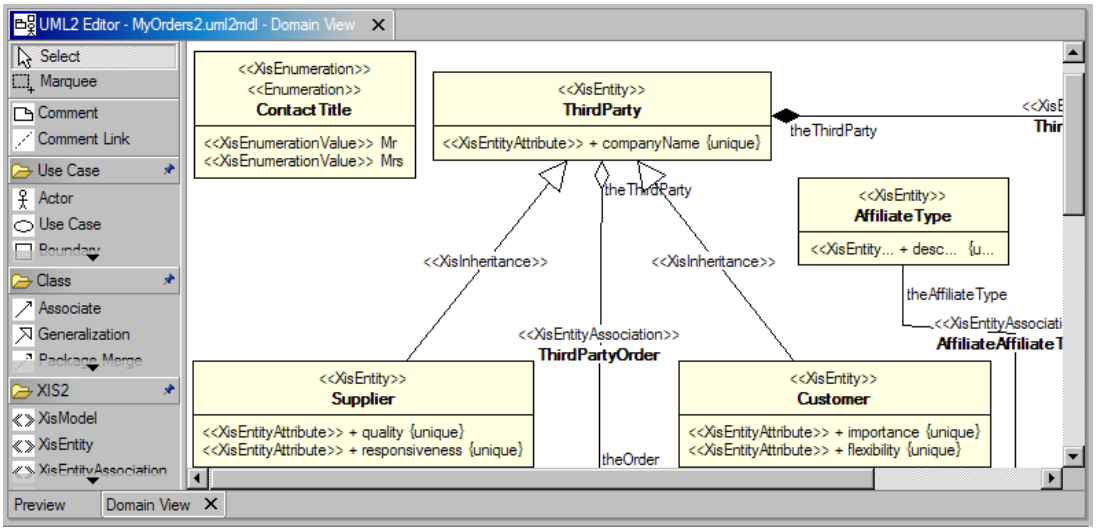

Figure 5.7: Screenshot of the "MyOrders2" Domain View with the XIS2 profile.

5.4 Defining and Executing "Model-to-Model" Transformations

Besides the definition of the profile's `Stereotypes`, XIS2 also includes a "model-to-model" transformation, the generation of the UserInterfaces View from the Entities View and the Use-Cases View, that is paramount to its "smart" design approach.

This transformation is implemented through C# code that performs the following tasks: (1) receive a `ModelContents`; (2) parse the `ModelContents` and build a `XisModel` that contains all the information about the model (obtained by the applications of the XIS2 stereotypes to UML elements); (3) generate the `XisUserInterfacesView` that corresponds to the Entities View and Use-Cases View; and (4) generate the UML elements, diagrams and visual information that correspond to the `XisUserInterfacesView` obtained in the previous step.

Listing 5.1 provides an overview of the `GenerateUserInterfacesView` class, which implements this "model-to-model" transformation (most of the source-code has been removed for text simplicity). The listing contains some particular points that are worth mentioning: (1) the two constants at the beginning of the file, `VERTICAL_SPACING` and `HORIZONTAL_SPACING`, determine the spacing between the visual elements of the diagrams that will be generated; (2) the `TransformModel` method is responsible for invoking the methods that process the received `ModelContents`, create the corresponding `XisModel`, generate the UserInterfaces View, and finally transform the result to UML; (3) the `umlModel` and the `xisModel` variables defined in the `TransformModel` method act as **Data Transfer Objects (DTO)** [Fowler 03], and are accessed and manipulated by most of the methods in the class; and (4) the `TransformUserInterfacesViewToUML` method adds a `RootNode` and two `Package`s corresponding to the NavigationSpace View and the InteractionSpace View, and invokes the methods that transform the `XisNavigationSpaceView` and the `XisInteractionSpaceView` to the corresponding UML elements and diagrams.

Listing 5.1: Overview of the source-code that implements the XIS2 "smart" approach transformation.

```
1  namespace Xis2.ModelTransformations {
2      public class GenerateUserInterfacesView {
3          private const int VERTICAL_SPACING = 150;
4          private const int HORIZONTAL_SPACING = 500;
5
6          public static bool ValidateTransformation(
7                  ModelContents model) { ... }
8
9          public static void TransformModel(ModelContents umlModel) {
10             // Perform validation (to ensure good model conversion...)
11             if(!ValidateTransformation(umlModel)) {
12                 return;
13             }
14
15             XisModel xisModel = new XisModel();
16
17             GetXisModel(umlModel, xisModel);
```

```
18            GetXisEntitiesView(umlModel, xisModel);
19            GetXisUseCasesView(umlModel, xisModel);
20
21            GenerateUserInterfaces(xisModel);
22
23            TransformUserInterfacesViewToUML(xisModel, umlModel);
24        }
25
26        protected static void TransformUserInterfacesViewToUML(
27                XisModel parent, ModelContents model) {
28            RootNode rootNode = UMLFactory.CreateRootNode();
29            rootNode.Name = "Generated Model";
30            model.AddRootNode(rootNode);
31
32            Package rootNodeView = UMLFactory.CreatePackage();
33            rootNodeView.Name = "Generated UserInterfaces View";
34            rootNode.AddView(rootNodeView);
35
36            Package navigationSpacesView = UMLFactory.CreatePackage();
37            navigationSpacesView.Name =
38                "Generated NavigationSpaces View";
39            rootNodeView.AddOwnedMember(navigationSpacesView);
40
41            Package interactionSpacesView = UMLFactory.CreatePackage();
42            interactionSpacesView.Name =
43                "Generated InteractionSpaces View";
44            rootNodeView.AddOwnedMember(interactionSpacesView);
45
46            XisUserInterfacesView uiView = parent.XisUserInterfacesView;
47            if(uiView == null) {
48                return;
49            }
50            XisNavigationSpaceView navView =
51                uiView.XisNavigationSpaceView;
52            XisInteractionSpaceView isView =
53                uiView.XisInteractionSpaceView;
54            if(navView == null || isView == null) {
55                return;
56            }
57
58            TransformInteractionSpacesViewToUML(
59                parent, model, interactionSpacesView);
60            TransformNavigationSpaceViewToUML(
61                parent, model, navigationSpacesView);
62        }
```

```
63      }
64  }
```

Although a great majority of this source-code is already implemented and functional, there are still some transformation methods and issues that need to be improved; this is why the ProjectIT-Studio still does not offer total support for the XIS2 "smart" approach.

Chapter 6

Conclusions and Future Work

This chapter is dedicated to the presentation of overall considerations about how this work was conducted and how it has evolved during its development. It is divided in three sections: (1) a discussion of the work developed (including the innovative features of UMLModeler relatively to other UML modeling tools) and improvements that could have been made during the software development process; (2) the planned future work for UMLModeler; and (3) the final conclusions that were originated by this work and the experience of developing in an environment like ProjectIT-Studio.

6.1 Discussion

The UMLModeler is a fundamental component of the ProjectIT-Studio in the context of the ProjectIT approach, because it is its responsibility to: (1) provide a smooth transition from requirements specification to source-code generation; and (2) provide a graphical UML modeling environment that is powerful, but also simple enough so that it does not overwhelm new users.

This plugin offers a range of features that can usually be found in UML modeling tools, and it is expected that many more features are added in future versions. Table 6.1 presents a comparison between the tools analyzed in "State of the Art", Chapter 2, and the UMLModeler.

It should be noted that, although some typical features have not yet been developed (such as "Model Pattern" capture and application), most of those features can easily be implemented, because of the modular architecture of UMLModeler and the "model manipulation by external plugins" functionality that UMLModeler provides.

An aspect that differentiates UMLModeler from typical UML modeling tools is its support for the graphical definition and application of an UML profile, which is inspired on the widely-accepted mechanism provided by Enterprise Architect [SparxSystems]. The

Features \ Tools	ArgoUML	Enterprise Architect	Poseidon for UML	Rational Rose	UMLModeler
Copy Diagram To Clipboard	No	Yes	Yes	Yes	Yes
Save Diagram As File	Yes	Yes	Yes	Yes (through Web Publishing)	Yes
Create Pattern	No	Yes	No	No	No
Create Classes From Patterns	No	Yes	No	Yes	No
Provides "Model Overview" Tree	Yes	Yes	Yes	Yes	Yes
Create Diagram Elements From Model Overview Tree	Yes	Yes	Yes	Yes	Yes
Create Profile	No	Yes	Yes (by creating a Plugin)	No	Yes
Create Stereotype	Yes	Yes	Yes	Yes	Yes
Custom Icons For Stereotypes	No	Yes	No	Yes (through .ini file)	Yes
Enforces User-Defined Constraints	Yes, using OCL (only for Classes and Features)	No	No	No	No
Supports UML 2.0	No	Yes	Yes	No	Yes
Supports User-Defined Model-to-Model Transformations	No	No	No	No	Yes
UML Standard Stereotypes	Yes	Yes	Yes	Yes	No
XMI Import/Export	Yes	Yes	Yes	Yes (through UniSys plugin)	No

Table 6.1: Comparison between UMLModeler and the UML tools previously analyzed.

most important changes, introduced by this work, to the Profile mechanism defined by the UML Superstructure [OMG 05b] were: (1) the `StereotypeApplication`, to overcome the fact that [OMG 05b] does not provide a concrete specification of the relationship between an UML element and a Stereotype that is applied to it; and (2) the `ProfilePackage`, to enable the segmentation of a `Profile` in various smaller packages, allowing the use of modeling approaches such as the "multi-view" approach employed by the XIS2 UML profile. Another advantage of the implemented mechanism is that profiles can be easily exchanged between different instances of ProjectIT-Studio, simply by the copying of the file that contains the profile's definition.

Another aspect that differentiates UMLModeler from other tools is its support for model manipulation by external entities. This mechanism allows developers to easily provide transformation operations, which can process and manipulate UML models for any number of purposes (e.g., the generation of the UserInterfaces View, in the case of XIS2 [Silva 07]); it also allows developers to provide validation code, which contributes to avoid potential errors in transformations and make the development of a model a dynamic process (because transformations only become available after certain conditions are met by the UML model). However, this mechanism could also be simpler, by providing a set of "model-to-model" transformation primitives to be used by the transformation operations

defined by developers. In fact, the "Generate UserInterfaces View" model transformation operation defined by XIS2 is currently implemented in C#, and its source-code consists of a few hundreds of lines.

Finally, the UMLModeler plugin should not be evaluated only by the functionalities it provides by itself, but also by its contribution to the ProjectIT-Studio and the ProjectIT approach. Besides the typical features previously mentioned, the UMLModeler also provides the added value of the integration with the Requirements and the MDDGenerator plugins [Silva 06a], allowing users to quickly obtain a model from the requirements specification, refine and transform that model, and finally generate the corresponding source-code automatically. In fact, in the case of ProjectIT-Studio, it is correct to say that the whole is worth much more than the sum of the parts.

6.2 Future Work

Although UMLModeler is currently stable and can be used in the context of the ProjectIT approach, it still lacks many of the features that users may consider important or even paramount.

A very important aspect that must be implemented soon is the set of UML Superstructure packages [OMG 05b] that have not yet been implemented in UMLModel. Although this UML metamodel implementation currently provides all the UML elements necessary for the XIS2 UML profile, it is necessary that the yet-undeveloped UML Superstructure packages and functionalities of the UMLModel be implemented in the near future, because of the following reasons: (1) the ProjectIT approach is language-independent, and thus the UMLModeler should also support any other UML profile; and (2) it is expected that the XIS2 UML profile definition grows in the near future, as user feedback is received and new avenues of research are pursued. Additionally, since the UML modeling plugin should support the entire UML Superstructure, the functionalities of UMLModeler must also be extended to provide this support.

The *serialization* functionalities of the UML model should also be improved. Although there is nothing wrong with the current strategy (save a model to an XML file, and use the correct deserializer to load a model from an XML file), it forces UMLModeler to work only with XML files. Since ProjectIT-Studio is supposed to communicate with other sources, such as ProjectIT-CommonDB and ProjectIT-LocalDB, the Strategy design pattern should also be used by the Serialization package to enable the choice of *where to serialize the model* (e.g., to an XMI file, to ProjectIT-CommonDB, to ProjectIT-LocalDB, or to any other locations that may be supported by ProjectIT-Studio in the future).

Another aspect to implement in the near future is the *support for constraint specification* (preferably using a standard language such as OCL [OMG 06a]), and the *validation of models based on those constraints*. UMLModeler should allow users to specify the following types of constraints: (1) *model-based constraints*, which are used to validate only the model in which they are specified; and (2) *profile-based constraints*, which are specified in the context of a profile, and are used to validate any models to which the profile is applied. The strategy currently planned for the implementation of this feature requires that each instance of the UMLModeler editor have a background thread that continuously validates the current model according to the constraints that are applied to the model. Obviously, this thread should have the lowest priority possible, so that the continuous validation of the model by the thread does not affect the performance of the ProjectIT-Studio's user-interface. Work on this feature has already begun, but it is still in its early stages and thus not yet available for user testing.

The *"model manipulation by external plugins" mechanism* should also be improved. One of the potential problems of this mechanism is that it allows each extension to specify *only one validation method* (which is used by UMLModeler when displaying the context menu of the Content Outline); if this validation method involves complex verifications, the performance of UMLModeler could be affected whenever the user tried to access the context menu. To solve this problem, the "UMLModelProvider" extension point will allow extensions to specify: (1) a **manipulation method**, which corresponds to the current *transformation method* of the extension point; (2) a method, called **quick validation method**, which should perform only a quick validation of the model, and will be used by UMLModeler to swiftly determine whether the action should be displayed in the context menu; and (3) any number of additional methods, called **regular validation methods**, that would be invoked only after the user selected the action, and which should perform model validations (with any level of complexity) before the manipulation method is invoked.

Another aspect that must be addressed is the *specification of "model-to-model" transformations*. Although these transformations can be applied through the use of the "UMLModelProvider" extension point, the only way to currently specify them is by using source-code. It would be desirable to define a mechanism that provides a set of *"model-to-model" transformation primitives*, which could then be used in the specification of "model-to-model" transformations at a high-level, with any level of complexity. The implementation of the support for these transformation primitives (and the transformations derived from those primitives) would likely be trivial, but the real challenge would be to define a set of primitives that successfully accomplishes the following requirements: (1) extensible enough to allow the definition of most (if not all) types of UML model trans-

formations; and (2) simple enough that users are not immediately confused when looking at a transformation specification. Because of this theoretical obstacle, this specification mechanism has not yet been defined or implemented.

Finally, *model patterns* are a functionality that typical users consider useful, but which was not implemented in UMLModeler due to time constraints. The application of such patterns could easily be accomplished by using the "UMLModelProvider" extension point to add the necessary UML elements to the current model. Nevertheless, support for model patterns of a more complex type (e.g., creating a file to store such patterns, editing a pattern file) would also be a welcome feature, but it will require some additional development efforts to extend the UMLModeler plugin accordingly.

6.3 Conclusions

The ProjectIT research project started in mid-2004, with the development of the ProjectIT-Enterprise and ProjectIT-Studio prototypes in previous works. Although the work on the ProjectIT-Studio resulted in the development of a stable integration platform to which functionalities of any kind could be added [Saraiva 05b], it was not until mid-2005 that development truly began on ProjectIT-Studio and its goal to streamline the software development process through the ProjectIT approach. This development resulted in the following plugins for ProjectIT-Studio: (1) Requirements [Ferreira 06]; (2) UMLModeler; and (3) MDDGenerator [Silva 06b].

Besides the creation of the ProjectIT-Studio plugins, the development efforts of this last year also brought forth the XIS2 UML profile (an evolution of the XIS profile), oriented towards the specification of "domain elements" with any level of granularity, the capture of interaction patterns, and the specification of user-interfaces through sketching techniques [Silva 07]. This profile proved fundamental for testing and evaluating the ProjectIT-Studio (and UMLModeler) as a platform to support the development of an interactive software system in a language-independent approach.

This thesis presented the development of the ProjectIT-Studio/UMLModeler, a plugin for ProjectIT-Studio that supports UML graphical modeling in the context of the ProjectIT approach. This plugin provides a graphical UML diagram editor and an easy-to-extend framework that allows the editing of any aspect of an UML model. It also offers a range of features that can usually be found in UML modeling tools, and it is expected that many more features are added in future versions.

Relatively to most UML modeling tools currently available, the UMLModeler also provides a number of innovative features, namely: (1) a mechanism to allow the easy definition and application of profiles and stereotypes, loosely based on the mechanism

provided by Enterprise Architect [SparxSystems], but with a higher degree of alignment with the UML Superstructure specification [OMG 05b]; and (2) the support for model manipulation by other plugins, which allows developers to easily add model manipulation operations (such as "model-to-model" transformations) that can be used for any number of purposes. Although these features can (and will) be improved in the future, they are already a noteworthy contribution to differentiate UMLModeler from other UML modeling tools.

UMLModeler is also integrated with the other ProjectIT-Studio plugins (Requirements and MDDGenerator). This integration with the ProjectIT-Studio plugins allows users to go from requirements specification to source-code generation in a smooth and efficient manner.

Finally, on a personal note, it should be mentioned that the development of the UMLModeler plugin was a very interesting experience, from both the professional and personal perspectives. The intense discussions and exchanges of ideas, which are a natural result from the work environment created by high-quality developers like the colleagues that are a part of the ProjectIT-Studio development team, were paramount to the selection and prioritization of the features that should be implemented, and to making ProjectIT-Studio the tool that it currently is. It is my hope that this environment lives on, so that ProjectIT-Studio can reach its full potential as a tool to address the software development life-cycle in its entirety.

References

[Archer 01] Archer, Tom. *Inside C#*. Microsoft Press, 2001. ISBN 0-7356-1288-9.

[ArgoUML] ArgoUML Project home. URL: `http://argouml.tigris.org`, accessed on Monday 5 June, 2006.

[ASP.NET] ASP.NET Web: The Official Microsoft ASP.NET Site: Home Page. URL: `http://www.asp.net/`, accessed on Monday 5 June, 2006.

[Booch 04] Booch, Grady, Brown, Alan, Iyengar, Sridhar, Rumbaugh, James, and Selic, Bran. *The MDA Journal: Model Driven Architecture Straight from the Masters*, chapter 11. An MDA Manifesto. Meghan-Kiffer Press, 2004. ISBN 0929652258. URL: `http://www.metamodel.com/wisme-2002/papers/atkinson.pdf`, accessed on Thursday 15 June, 2006.

[Buchanan 02] Buchanan, Richard D. and Soley, Richard Mark. An OMG Whitepaper: Aligning Enterprise Architecture and IT Investments with Corporate Goals, 2002. URL: `http://www.bptrends.com/publicationfiles/METAOMGWP1-15-03.pdf`, accessed on Monday 5 June, 2006.

[DSMForum] DSM Forum: Domain-Specific Modeling. URL: `http://www.dsmforum.org/`, accessed on Monday 5 June, 2006.

[Eclipse] Eclipse.org Main page. URL: `http://www.eclipse.org`, accessed on Monday 5 June, 2006.

[EclipseEMF] Eclipse Tools – EMF Home. URL: `http://www.eclipse.org/emf`, accessed on Monday 5 June, 2006.

[EclipseGEF a] eclipsewiki – GefDescription. URL: `http://eclipsewiki.editme.com/GefDescription`, accessed on Monday 16 October, 2006.

[EclipseGEF b] The Eclipse Graphical Editing Framework Project. URL: `http://www.eclipse.org/gef`, accessed on Monday 5 June, 2006.

[Eclipse.NET] SourceForge.net: Eclipse.NET. URL: http://sourceforge.net/projects/eclipsedotnet, accessed on Friday 13 October, 2006.

[EclipseSWT] SWT: The Standard Widget Toolkit. URL: http://www.eclipse.org/swt, accessed on Thursday 19 October, 2006.

[EclipseUML2] The Eclipse UML2 Project. URL: http://www.eclipse.org/uml2, accessed on Monday 5 June, 2006.

[Ferreira 06] Ferreira, David de Almeida. ProjectIT-RSL. Relatório Final de Trabalho Final de Curso, Instituto Superior Técnico, Portugal, October 2006.

[Fowler 03] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003. ISBN 0-321-12742-0.

[France 06] France, Robert B., Ghosh, Sudipto, Dinh-Trong, Trung, and Solberg, Arnor. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer*, 39(2):59–66, February 2006. ISSN 0018-9162. URL: http://doi.ieeecomputersociety.org/10.1109/MC.2006.65, accessed on Monday 5 June, 2006.

[Gamma 95] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[Gamma 03] Gamma, Erich and Beck, Kent. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. Addison-Wesley, October 2003. ISBN 0-321-20575-8.

[Gentleware] Gentleware - model to business: poseidon for uml. URL: http://www.gentleware.com/products.html, accessed on Sunday 17 December, 2006.

[IBM] IBM. IBM Software – Rational Rose Data Modeler – Product Overview. URL: http://www-306.ibm.com/software/awdtools/developer/datamodeler/, accessed on Monday 5 June, 2006.

[IKVM.NET] IKVM.NET Home Page. URL: http://www.ikvm.net, accessed on Monday 5 June, 2006.

[INESC-ID] INESC-ID, Information Systems Group (GSI). URL: ProjectIT Website: http://berlin.inesc-id.pt/alb/ProjectIT@81.aspx, accessed on Monday 11 December, 2006.

[Java] Java Technology. URL: http://java.sun.com, accessed on Monday 5 June, 2006.

[Kelly 05] Kelly, Steven. Improving Developer Productivity With Domain-Specific Modeling Languages. *Developer.* – The Independent Magazine for Software Developers*, July 2005. URL: http://www.developerdotstar.com/mag/articles/domain_modeling_language.html, accessed on Friday 16 June, 2006.

[Kleppe 03] Kleppe, Anneke, Warmer, Jos, and Bast, Wim. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, April 2003. ISBN 0-321-19442-X.

[MartinFowler] Martin Fowler. URL: http://www.martinfowler.com, accessed on Wednesday 7 June, 2006.

[MDA] OMG Model Driven Architecture. URL: http://www.omg.org/mda, accessed on Monday 5 June, 2006.

[Mittal 05] Mittal, Kunal. Introducing Rational Software Modeler, November 2005. URL: http://www-128.ibm.com/developerworks/rational/library/05/329_kunal/, accessed on Monday 18 December, 2006.

[ModelWare] Model transformation at Inria / Introduction to Model-Driven Engineering. URL: http://modelware.inria.fr/article65.html, accessed on Saturday 24 June, 2006.

[MOF] OMG's MetaObject Facility (MOF) Home Page. URL: http://www.omg.org/mof/, accessed on Monday 5 June, 2006.

[.NET a] Microsoft .NET Homepage. URL: http://www.microsoft.com/net/, accessed on Monday 5 June, 2006.

[.NET b] MSDN .NET Framework Developer Center: Technology Overview. URL: http://msdn2.microsoft.com/en-us/netframework/aa497336.aspx, accessed on Saturday 16 December, 2006.

[.NET c] Overview of the .NET Framework. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpovrintroductiontonetframeworksdk.asp, accessed on Monday 5 June, 2006.

[.NET d] What's New in the .NET Framework Version 2.0. URL: http://msdn2.microsoft.com/en-us/library/t357fb32.aspx, accessed on Friday 13 October, 2006.

[Nóbrega 06] Nóbrega, Leonel, Nunes, Nuno Jardim, and Coelho, Helder. The Meta Sketch Editor: a Reflexive Modeling Editor. In G. Calvary, C. Pribeanu, G. Santucci, and J. Vanderdonckt, editors, *Computer-Aided Design of User Interfaces V – Proceedings of the 6th International Conference on Computer-Aided Design of User Interfaces (CADUI'2006)*, pages 199–212. Springer-Verlag, Berlin, Germany, June 2006.

[nUML] Main Page - nUML. URL: http://sourceforge.net/projects/numl, accessed on Sunday 22 October, 2006.

[OMG] Object Management Group. URL: http://www.omg.org, accessed on Monday 5 June, 2006.

[OMG 05a] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. URL: http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf, accessed on Thursday 22 June, 2006, November 2005.

[OMG 05b] Object Management Group. Unified Modeling Language: Superstructure – Specification Version 2.0, August 2005. URL: http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf, accessed on Monday 29 May, 2006.

[OMG 06a] Object Management Group. Object Constraint Language – Specification Version 2.0, May 2006. URL: http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf, accessed on Friday 16 June, 2006.

[OMG 06b] Object Management Group. Unified Modeling Language: Diagram Interchange – Specification Version 1.0, April 2006. URL: http://www.omg.org/cgi-bin/apps/doc?formal/06-04-04.pdf, accessed on Friday 16 June, 2006.

[OMG 06c] Object Management Group. Unified Modeling Language: Infrastructure – Specification Version 2.0, March 2006. URL: http://www.omg.org/cgi-bin/apps/doc?formal/05-07-05.pdf, accessed on Monday 29 May, 2006.

[Saraiva 05a] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. Eclipse.NET: An Integration Platform for ProjectIT-Studio. In *Proceedings of the First International Conference of Innovative Views of .NET Technologies (IVNET'05)*, pages 57–69. Instituto Superior de Engenharia do Porto and Microsoft, July 2005. ISBN 972-8688-31-8. URL: `http://w2ks.dei.isep.ipp.pt/labdotnet/files/IVNET/EclipseNet_p.pdf`, accessed on Wednesday 21 June, 2006.

[Saraiva 05b] Saraiva, João Paulo Pedro Mendes de Sousa. Desenvolvimento Automático de Sistemas. Relatório Final de Trabalho Final de Curso, Instituto Superior Técnico, Portugal, July 2005.

[Schmidt 06] Schmidt, Douglas C. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, February 2006. URL: `http://doi.ieeecomputersociety.org/10.1109/MC.2006.58`, accessed on Monday 5 June, 2006.

[Sellers 05] Henderson-Sellers, Brian. UML the Good, the Bad or the Ugly? Perspectives from a panel of experts. *Software and Systems Modeling*, 4(1):4–13, February 2005. ISSN 1619-1366. URL: `http://dx.doi.org/10.1007/s10270-004-0076-8`, accessed on Monday 5 June, 2006.

[Silva 01] Silva, Alberto and Videira, Carlos. *UML, Metodologias e Ferramentas CASE*. Centro Atlântico, Portugal, April 2001. ISBN 972-8426-36-4.

[Silva 03a] Silva, Alberto Rodrigues da. The XIS Approach and Principles. In *Proceedings of the 29th EUROMICRO Conference*. IEEE Computer Society, September 2003. URL: `http://doi.ieeecomputersociety.org/10.1109/EURMIC.2003.1231564`, accessed on Wednesday 14 June, 2006.

[Silva 03b] Silva, Alberto Rodrigues da, Lemos, Gonçalo, Matias, Tiago, and Costa, Marco. The XIS Generative Programming Techniques. In *Proceedings fo the 27th COMPSAC Conference*. IEEE Computer Society, November 2003. URL: `http://doi.ieeecomputersociety.org/10.1109/CMPSAC.2003.1245347`, accessed on Monday 5 June, 2006.

[Silva 04] Silva, Alberto Rodrigues da. O Programa de Investigação ProjectIT (whitepaper), October 2004. URL: `http://berlin.inesc-id.pt/alb/uploads/1/193/pit-white-paper-v1.0.pdf`, accessed on Monday 5 June, 2006.

[Silva 05a] Silva, Alberto and Videira, Carlos. *UML, Metodologias e Ferramentas CASE, 2ª Edição, Volume 1.* Centro Atlântico, Portugal, May 2005. ISBN 989-615-009-5.

[Silva 05b] Silva, Alberto Rodrigues da. Programa de I&D ProjectIT, April 2005. URL: `http://berlin.inesc-id.pt/alb/uploads/1/438/pit-status-2005-v0.1.pdf`, accessed on Monday 5 June, 2006.

[Silva 06a] Silva, Alberto, Videira, Carlos, Saraiva, João, Ferreira, David, and Silva, Rui. The ProjectIT-Studio, an integrated environment for the development of information systems. In *Proceedings of the Second International Conference of Innovative Views of .NET Technologies (IVNET'06)*, pages 85–103. Sociedade Brasileira de Computação and Microsoft, October 2006. ISBN 85-99580-02-7. URL: `http://berlin.inesc-id.pt/alb/static/papers/2006/ivnet2006-pit-v1.0c.pdf`, accessed on Thursday 14 September, 2006.

[Silva 06b] Silva, Rui Miguel Tavares da. ProjectIT – Produção Automática de Software. Relatório Final de Trabalho Final de Curso, Instituto Superior Técnico, Portugal, October 2006.

[Silva 07] Silva, Alberto, Saraiva, João, Silva, Rui, and Martins, Carlos. XIS – UML Profile for eXtreme Modeling Interactive Systems. In *Fourth International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007) (to appear)*. IEEE Computer Society, March 2007.

[SparxSystems] Systems, Sparx. Enterprise Architect – UML Design Tools and UML CASE tools for software development. URL: `http://www.sparxsystems.com/products/ea.html`, accessed on Monday 5 June, 2006.

[Thomas 04] Thomas, Dave. MDA: Revenge of the Modelers or UML Utopia? *IEEE Software*, 21(3):15–17, May/June 2004. ISSN 0740-7459. URL: `http://doi.ieeecomputersociety.org/10.1109/MS.2004.1293067`, accessed on Monday 5 June, 2006.

[UML] Object Management Group – UML. URL: `http://www.uml.org`, accessed on Monday 5 June, 2006.

[Videira 05] Videira, Carlos and Silva, Alberto Rodrigues da. Patterns and metamodel for a natural-language-based requirements specification language. In *Proceedings of CaiSE05 Forum*, pages 189–194. Faculdade de Engenharia da Universidade do Porto, June 2005. URL: `http://berlin.inesc-id.pt/alb/static/papers/2005/cv-caise2005.pdf`, accessed on Tuesday 10 October, 2006.

[XMI] Object Management Group. XML Metadata Interchange. URL: `http://www.omg.org/technology/documents/formal/xmi.htm`, accessed on Monday 5 June, 2006.

[XML a] Extensible Markup Language (XML). URL: `http://www.w3.org/XML`, accessed on Monday 5 June, 2006.

[XML b] XML DOM Tutorial. URL: `http://www.w3schools.com/dom/`, accessed on Tuesday 28 November, 2006.

Glossary

A

Action A possible operation, presented to the user when a right-mouse-button click occurs over the area which provides the Action, p. 62.

ArgoUML An open-source UML 1.4 modeling tool written in Java. It differentiates itself from other UML modeling tools by its use of "cognitive psychology" to detect model inconsistencies and promote modeling best-practices, p. 14.

C

Command Class used by GEF to implement the Command design pattern. A Command is used to encapsulate an user action as an object, providing the ability to support undoable operations in a GEF-based editor, p. 27.

Common Language Runtime (CLR) A language-neutral development and execution environment that provides services to help manage the execution of .NET-based applications, p. 20.

Content Outline A mechanism provided by Eclipse.NET for displaying outline information about a document. Any editor plugin can use this mechanism by providing an Outline Page. UMLModeler provides such an Outline Page, p. 52.

E

Eclipse Graphical Editing Framework (GEF) An open-source framework dedicated to providing a rich and consistent graphical editing environment, based on the Model-View-Controller pattern, for plugins on the Eclipse Platform, p. 25.

Eclipse UML2 An UML 2.0 implementation library, still in development, based on the Eclipse Modeling Framework. Its purpose is to provide an infrastructure that allows Eclipse plugins to create and manipulate UML models, p. 29.

Eclipse.NET An open-source project that aims to deliver an extensible, plugin-based platform to support the development of integrated tools for the .NET runtime environment, p. 23.

Eclipse.NET Extension Point A mechanism that allows Eclipse.NET plugins to interact among themselves. Developers provide "extensions" to a plugin's extension points in order to add (or change) functionalities to the plugin, p. 24.

Eclipse.NET Plugin Eclipse.NET's smallest unit of behavior. Any tool can be developed as a plugin or as a set of plugins, p. 24.

EditPart Class used by GEF to define the *Controller* component of the Model-View-Controller design pattern, p. 27.

Enterprise Architect (EA) A commercial UML 2.0 modeling tool that features good usability and a minimal learning curve. Currently known through the Software Engineering community as one of the most versatile UML modeling tools available, p. 15.

F

Figure The graphical building blocks of Draw2D. GEF's implementation of the MVC design pattern uses Figures to define the *View* component, p. 27.

G

Graphical Modeler The GEF-based UML diagram editor that is used by ProjectIT-Studio/UMLModeler, p. 55.

I

IKVM.NET A Java Virtual Machine for the .NET platform. Its goal is to reduce the gap between these two technological platforms by providing an interoperability mechanism, p. 22.

M

Microsoft Intermediate Language (MSIL) The language to which source-code written in any .NET-based language is initially compiled. MSIL is interpreted at runtime by the .NET Common Language Runtime, p. 20.

Microsoft .NET Framework (.NET) A platform, created by Microsoft, that supports the development and execution of command-line or form-based applications, web applications, and web services, p. 20.

Model In the ProjectIT approach, it is an abstract representation of a software system, created by the designer and based on a given UML profile (e.g., the XIS2 UML profile), p. 34.

Model-Driven Architecture (MDA) A framework for the software development life cycle, and its main characteristic is the importance of models in the development process. Based on other OMG standards such as UML, MOF, QVT, and XMI, p. 12.

N

nUML A .NET library for manipulating UML 2.0 models, still in its early stages, p. 29.

P

Palette The palette, displayed within the Graphical Modeler, displays a set of tools for creating objects in the diagram, p. 60.

Poseidon for UML A commercial UML 2.0 modeling tool based on ArgoUML, p. 16.

Preferences A mechanism provided by Eclipse.NET for displaying a window with a series of options that can affect any of the platform's functionalities. Any plugin can use this mechanism by providing a Preference Page. UMLModeler provides such a Preference Page, p. 68.

ProjectIT approach A platform-independent approach, based on MDA, for designing interactive software systems, p. 3.

ProjectIT-MDD The functional component of ProjectIT related to the area of information systems' modeling and MDE. Allows the specification of models, transformations between models defined in different languages, and the automatic generation of artifacts, p. 4.

ProjectIT-Studio An Eclipse.NET-based application that supports the ProjectIT approach. ProjectIT-Studio can also be seen as an orchestration of three different plugins: ProjectIT-Studio/Requirements, ProjectIT-Studio/UMLModeler and ProjectIT-Studio/MDDGenerator, p. 36.

ProjectIT-Studio/MDDGenerator The ProjectIT-Studio plugin responsible for the generation of system artifacts based on models, p. 34.

ProjectIT-Studio/Requirements The ProjectIT-Studio plugin responsible for requirements engineering issues, p. 33.

ProjectIT-Studio/UMLModeler The ProjectIT-Studio plugin responsible for standard UML modeling. Allows the creation of visual models using UML 2.0 and a given UML profile (e.g., the XIS2 UML profile), p. 33.

Property Form A window that provides all the details about a selected UML element. Usually presented when the user double-clicks over an UML element, p. 66.

R

Rational Rose A commercial UML 1.x modeling tool, well known in the Software Engineering community. Recently replaced by the Rational Software Modeler, in the scope of IBM's new common development environment, p. 17.

Request Class used by GEF to interact with EditParts. Requests are used in various tasks, such as targeting, filtering the selection, graphical feedback, and obtaining Commands, p. 27.

S

Software Architecture In the ProjectIT approach, it is a generic representation of a software platform, created by the Architect by developing templates for that target platform, p. 34.

Standard Widget Toolkit (SWT) A platform-specific runtime that offers widget objects to Java developers, but underneath uses native code calls to create and interact with those controls, making SWT-based applications essentially indistinguishable from native platform applications, p. 24.

T

Template In the ProjectIT approach, it is a generic representation of a software artifact that supports the "Model2Code" transformations, p. 34.

U

UMLModel ProjectIT-Studio's implementation of the UML 2.0 metamodel. Provides additional features, such as visual information, and a mechanism for profile definition and application, p. 41.

Unified Modeling Language (UML) A general-purpose modeling language, originally designed to specify, visualize, construct, and document information systems, p. 10.

W

Wizards A mechanism provided by Eclipse.NET for displaying a series of windows that guide the user through the performing of a task. Any plugin can use this mechanism by providing the appropriate extension. UMLModeler provides such an extension, p. 70.

X

"eXtreme modeling Interactive Systems" UML profile (XIS2) An UML profile oriented towards the development of interactive software systems. Its main goal is to allow the modeling of the various aspects of an interactive software system, in a way that should be as simple and efficient as possible, p. 81.

XML Metadata Interchange (XMI) A standard for exchanging, defining, interchanging, and manipulating metadata information by using XML. Commonly used as a format to exchange UML models between tools, p. 12.

Appendix A

The "MyOrders2" Case Study

This appendix presents the "MyOrders2" case study, which was created to evaluate the ProjectIT approach and its supporting tool, ProjectIT-Studio.

A.1 Introduction

MyOrders2 is a software application designed to manage suppliers, clients, products, and orders. This application is generic enough to be easily adaptable to any type of business. It is also purposely very limited (on purpose) regarding data-manipulation operations of a certain complexity.

MyOrders2 is a reference case study, created to test and evaluate the various plugins of the ProjectIT-Studio tool, namely: (1) ProjectIT-Studio/Requirements, for requirements specification; (2) ProjectIT-Studio/UMLModeler, for graphical specification of UML models; and (3) ProjectIT-Studio/MDDGenerator, for code generation.

A.2 Requirements

To better understand this case study, consider the example of a paper store. In a paper store, there are products (e.g., school books) that clients are interested in acquiring. When a client expresses that interest, a corresponding order is created listing the desired products. On the other hand, the store also creates supply orders when it needs to acquire products (e.g., when stocks run out, or when new products are released).

Given this example and observing other similar examples, the following requirements can be presented for the MyOrders2 software system:

1. The system must allow the management of clients and suppliers. An order can have any number of lines, and any line must provide information about the product, the desired quantity and the unitary price;
2. The system must allow the management of products that can be included in orders;
3. The system must allow the management of clients, which are related to a certain market;
4. The system must allow the management of markets;
5. The system must allow the management of suppliers, which provide a variety of products.

The system must also provide mechanisms for access control and data manipulation, which are presented in the following requirements:

1. Any user may login to the system, and any authenticated user may logout;
2. The administrator is a registered user that can create new users and assign them roles, which are then used to control access to data and available operations;
3. Guest users may view the product listing, and view the details about any product;
4. Registered users may view and change all the data that is managed by the system.

A.3 Design

The domain model presented in Figure A.1 was designed in order to support the requirements presented in the previous section. The model already presents the XIS2 UML profile's stereotypes applied to the entity classes.

In this domain model, `Suppliers` and `Customers` inherit from `ThirdParty`, which has a set of `Affiliates` and a set of `Orders` placed by that entity. An `Order` is composed by several lines (`OrderDetails`) that associate `Products` to prices and quantities. `Suppliers` have a set of `Products` that they supply; likewise, `Products` are supplied by a set of `Suppliers`.

To create `Customer Orders`, the application user must: (1) create a `Customer` account, if it does not yet exist, and the corresponding `Affiliates`; and (2) create a new `Order` and the respective `OrderDetails`, each of which must have a corresponding `Product` (which is selected from a list). The user must proceed in the same manner for `Supplier Orders`, although in this case the `Supplier` must be selected first.

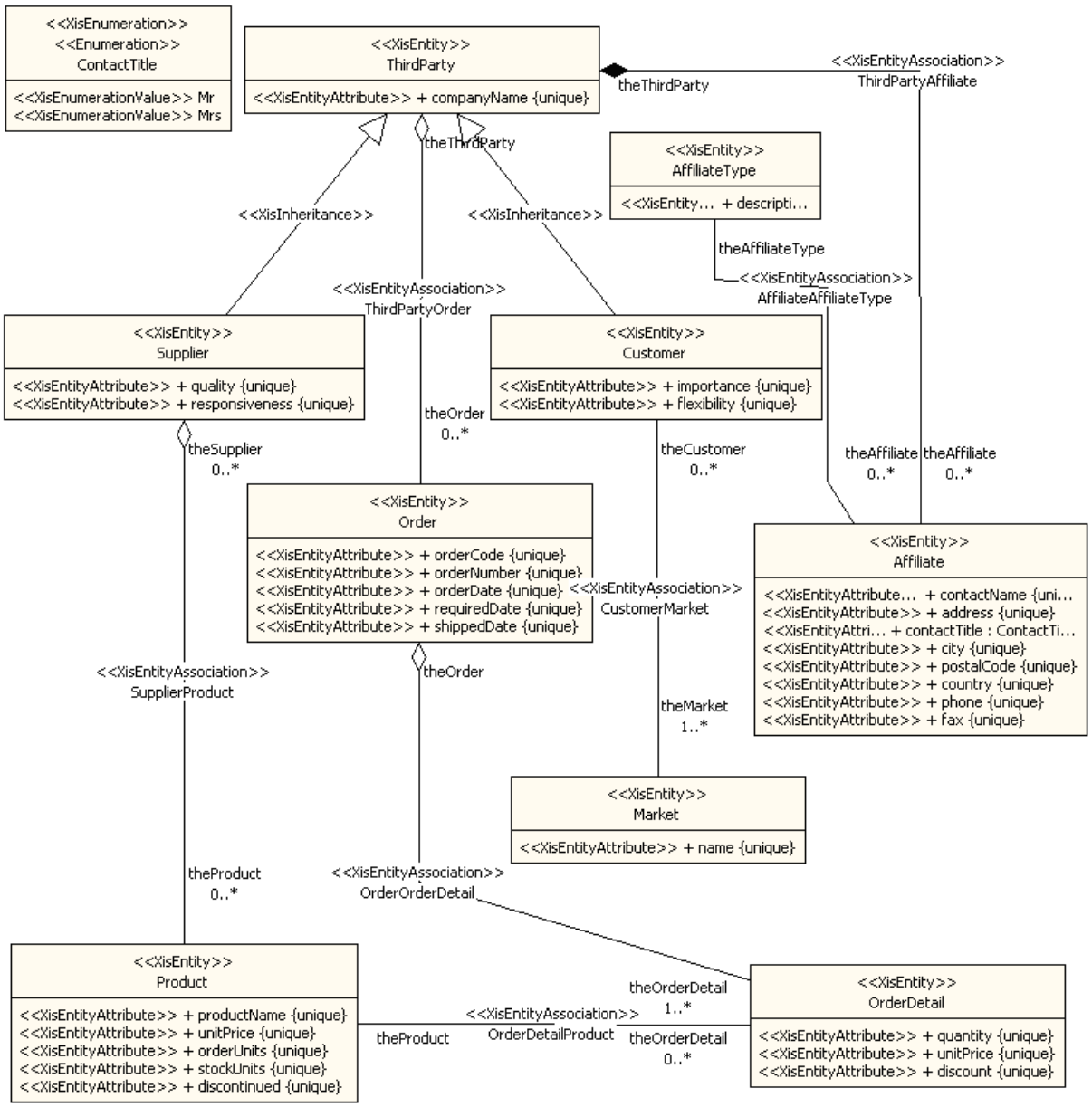

Figure A.1: Domain model of MyOrders.

The user may associate `Customers` to `Markets`, by doing one of the following actions: (1) selecting the `Customer` and associating it to the `Markets`; or (2) selecting the `Market` and associating it to the `Customers`. The user can also associate `Suppliers` to `Products` in a similar manner.

To support these operations, MyOrders2 must provide the user-interfaces and the navigation flow illustrated in Figure A.2.

Figures A.3, A.5, and A.7 presents some of the defined interaction spaces (not all interaction spaces are presented, to keep this description simple). Their mappings are also presented in Figures A.4, A.6, and A.8, respectively.

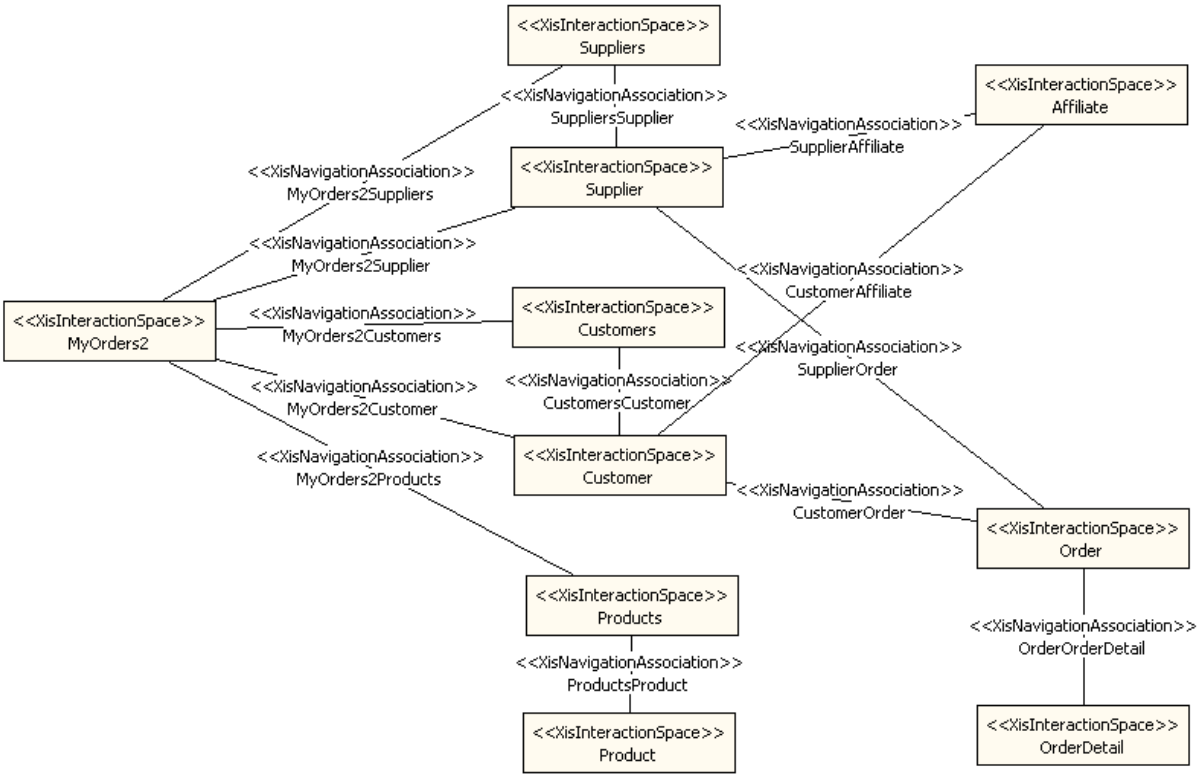

Figure A.2: NavigationSpace View of MyOrders2.

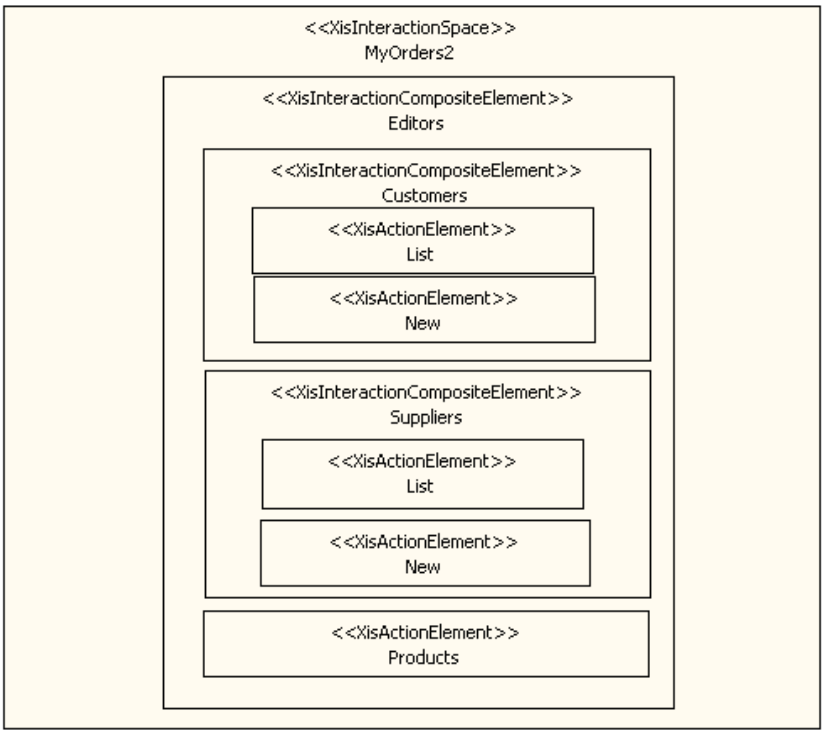

Figure A.3: Main interaction space.

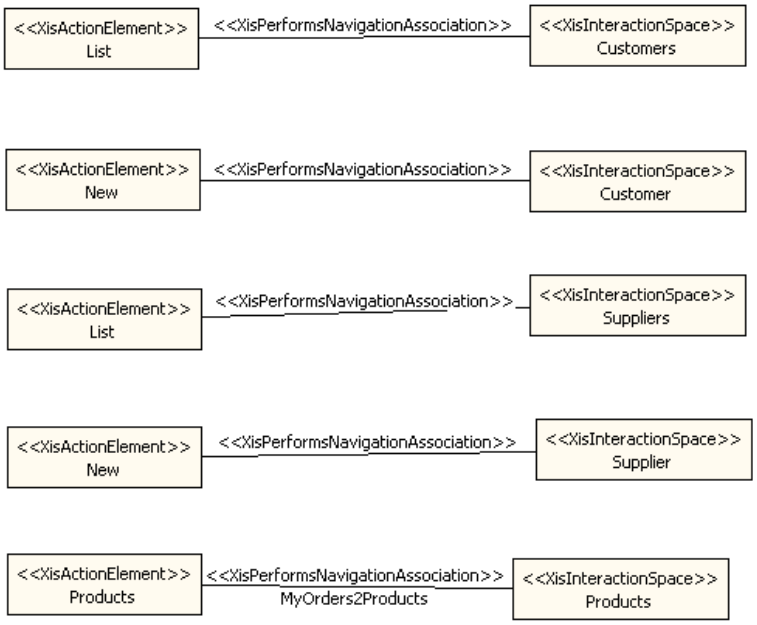

Figure A.4: Main interaction space mappings.

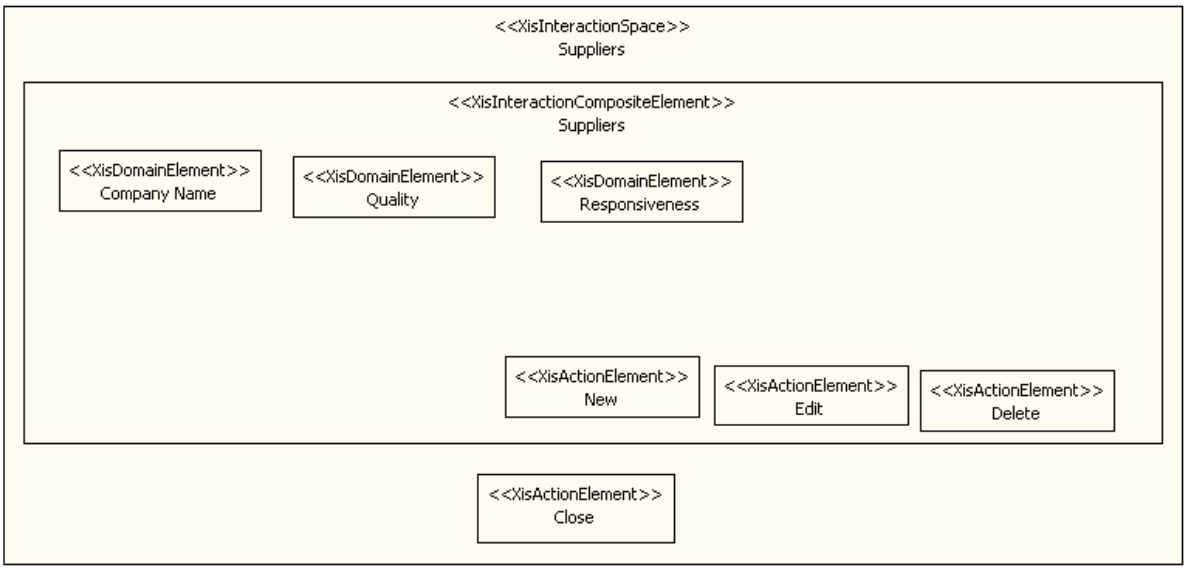

Figure A.5: Suppliers interaction space.

A.4 Development

The development of this application is done by following the ProjectIT approach:

- It begins with the Requirements Engineer, who specifies the requirements in ProjectIT-Studio/Requirements. Using the created document, the Requirements Engineer ex-

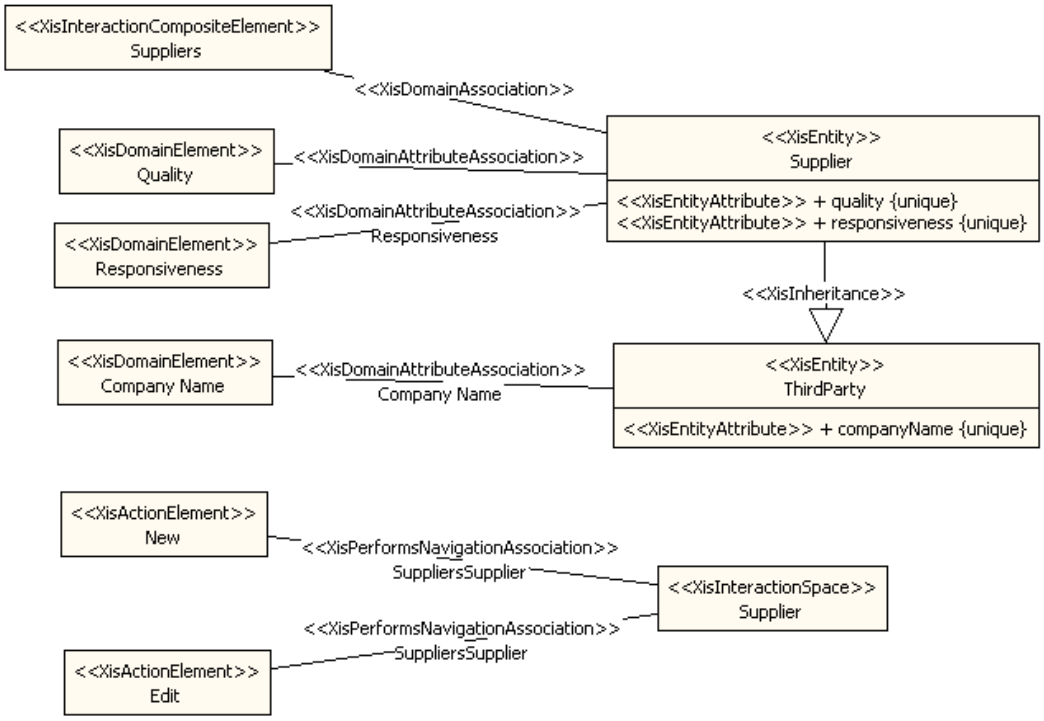

Figure A.6: Suppliers interaction space mappings.

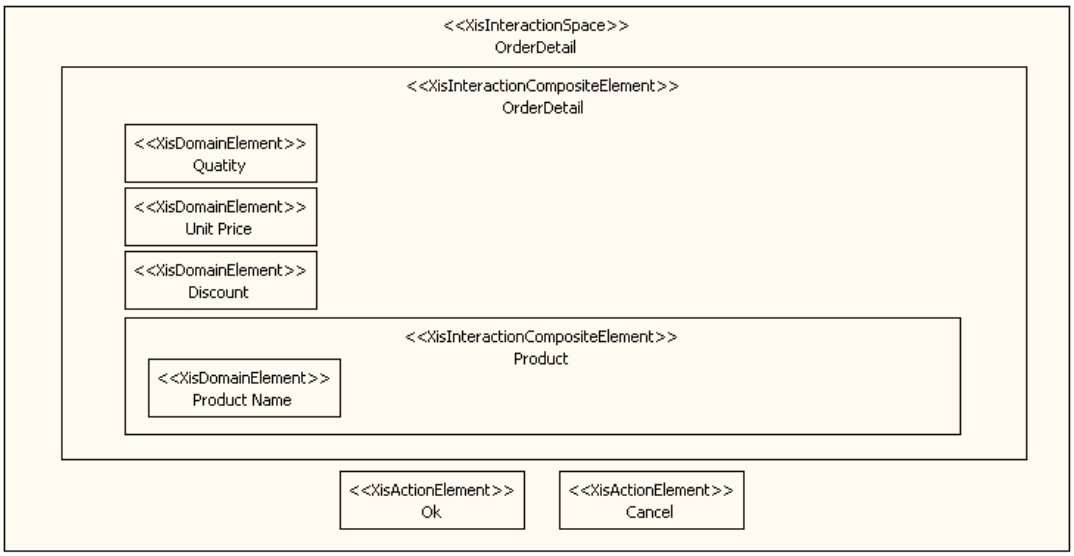

Figure A.7: OrderDetail interaction space.

ecutes the transformation that generates the corresponding UML model, containing system's domain model elements and its actors;

- The Designer then refines and completes the model in ProjectIT-Studio/UMLModeler, by adding diagrams that specify the system's the user-interfaces;

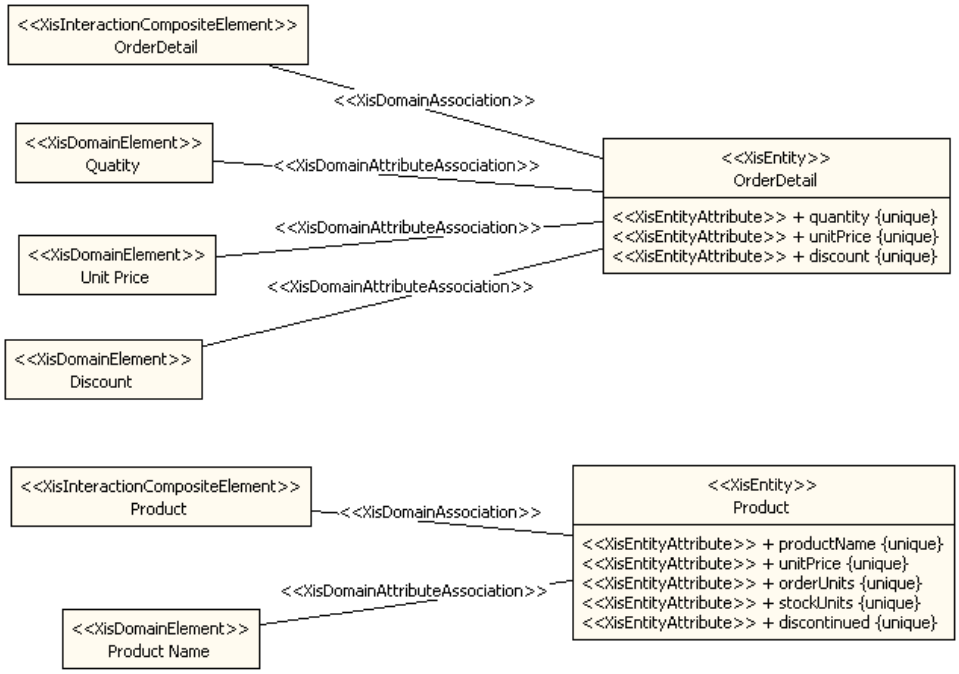

Figure A.8: OrderDetail interaction space mappings.

- Afterward, the Programmer generates the application's source-code by using ProjectIT-Studio/MDDGenerator, develops the requested features that are not yet supported by the XIS2 profile (by using a traditional IDE, such as Microsoft Visual Studio), and compiles the application;

- Finally, the Integrator creates the application database by using the generated SQL scripts, and configures the application's access to the database.

A.5 Results

Figure A.9 shows the deployment diagrams corresponding to the platforms for which the application was generated. The deployment architecture for the Windows Form platform is composed by a data repository and the application. On the other hand, the deployment architecture for the ASP.NET platform is composed by a data repository, a web-server, and a web-browser.

The component architecture of the application is the same, regardless of whether the application is generated for Windows Forms or for ASP.NET, and is composed by three components, which are shown in Figure A.10: (1) a component with classes that represent the information obtained from the database; (2) a component with classes for managing

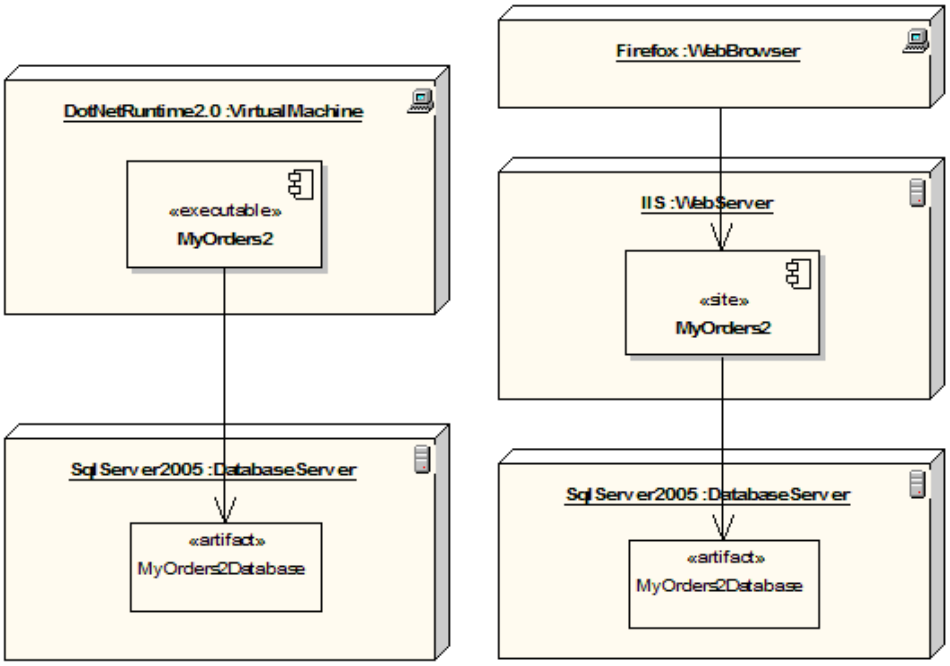

Figure A.9: Deployment diagrams of the generated application, for Windows Forms (left) and for ASP.NET (right).

access to the database; and (3) a component with user-interfaces for manipulating the application's data.

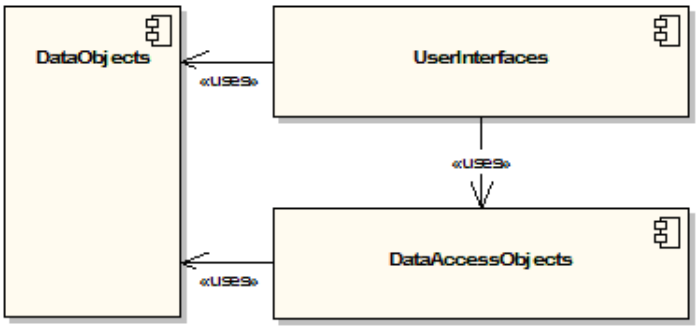

Figure A.10: Component diagram of the generated application.

A.6 Screens

This section presents some screens of the MyOrders2 application in the Windows Forms and the ASP.NET platforms. Figures A.11, A.13, and A.15 present some screens in the

Windows Forms platform. Figures A.12, A.14, and A.16 present the corresponding screens in the ASP.NET platform.

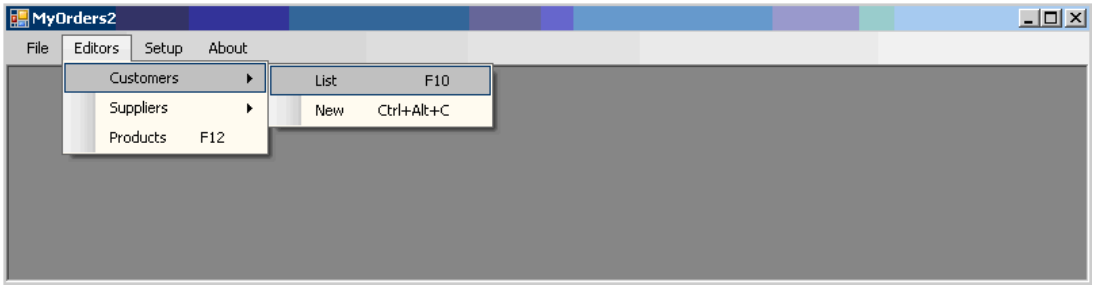

Figure A.11: Main screen (Windows Forms).

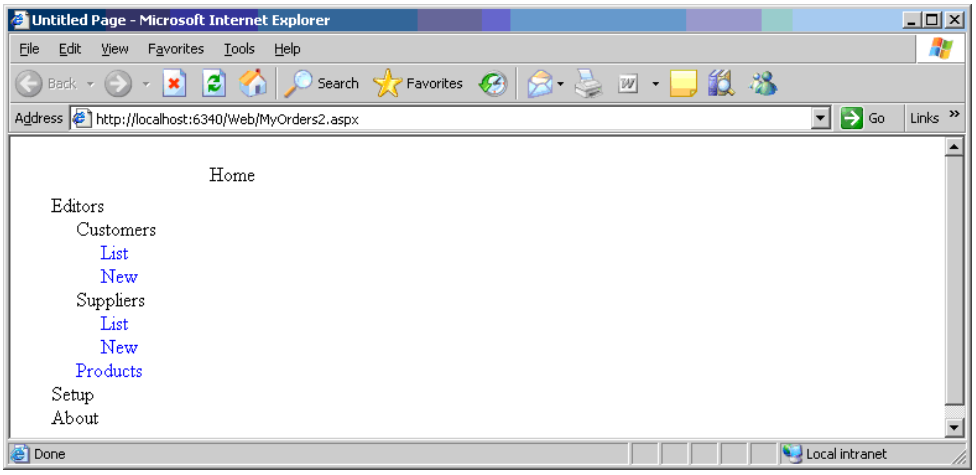

Figure A.12: Main screen (ASP.NET).

A.7 Conclusions

The generated application accomplishes the specified requirements. It also has the advantage of operating over two different platforms: Windows Forms and ASP.NET.

However, the application only provides basic management operations, such as: (1) creating, (2) editing, (3) viewing, or (4) deleting entities. Operations of a more complex nature, such as elaborate queries to the application's database, require the usage of external applications or changes to the generated source-code.

The ProjectIT approach, when compared with the traditional software development approaches, presents the following advantages: (1) lower time-to-market for the desired product; (2) the entire application's source-code and user-interfaces are consistent; (3) the developer's experience can be reused, by using the templates again in other projects;

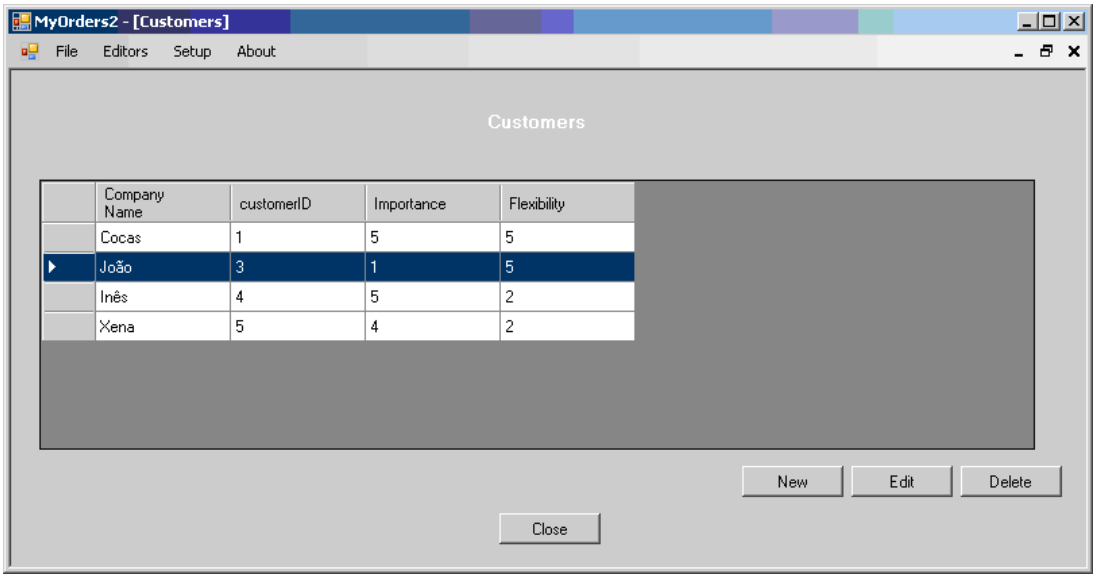

Figure A.13: Customers listing screen (Windows Forms).

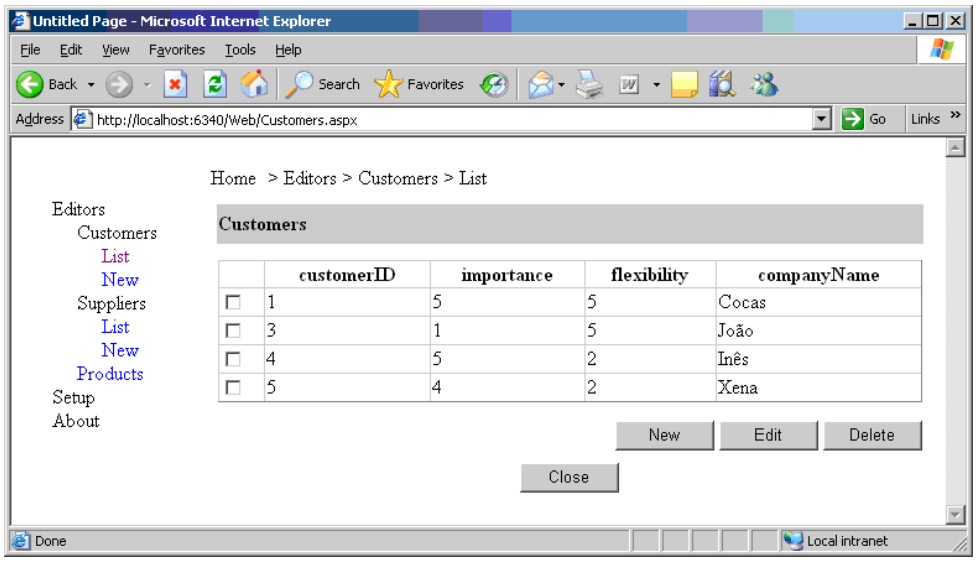

Figure A.14: Customers listing screen (ASP.NET).

and (4) the application's source-code can be generated for another target platform, by changing the software architecture. However, the development of the system's model and the software architecture's templates can be a time-consuming task.

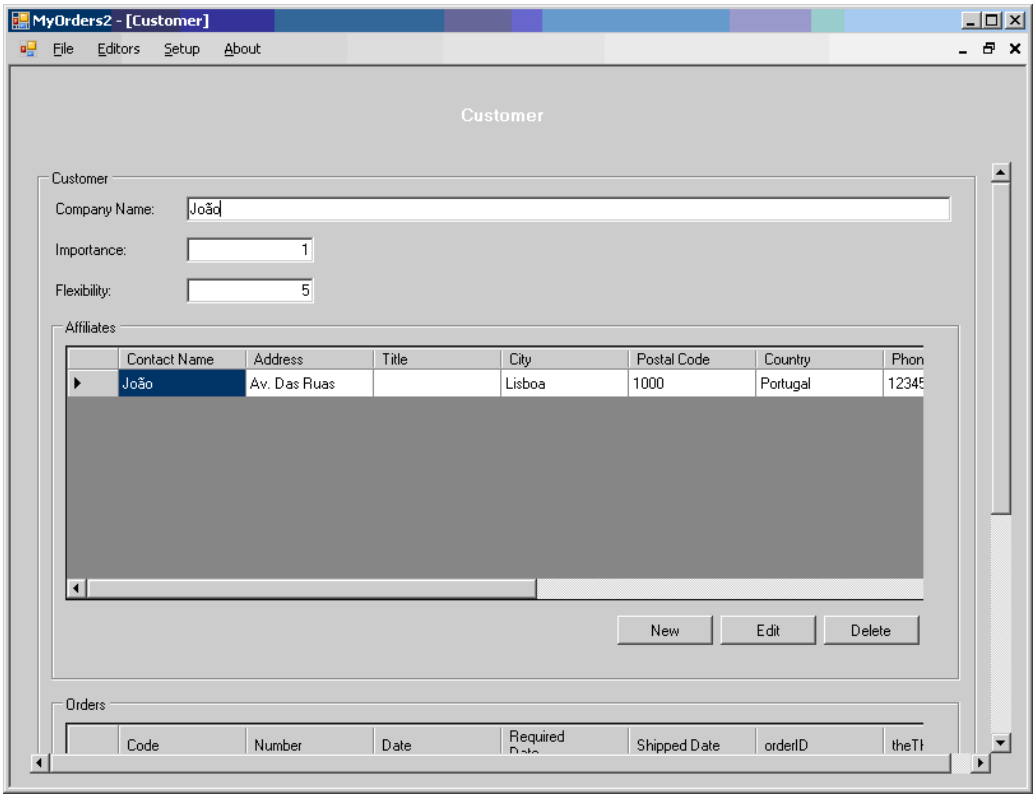

Figure A.15: Customer editing screen (Windows Forms).

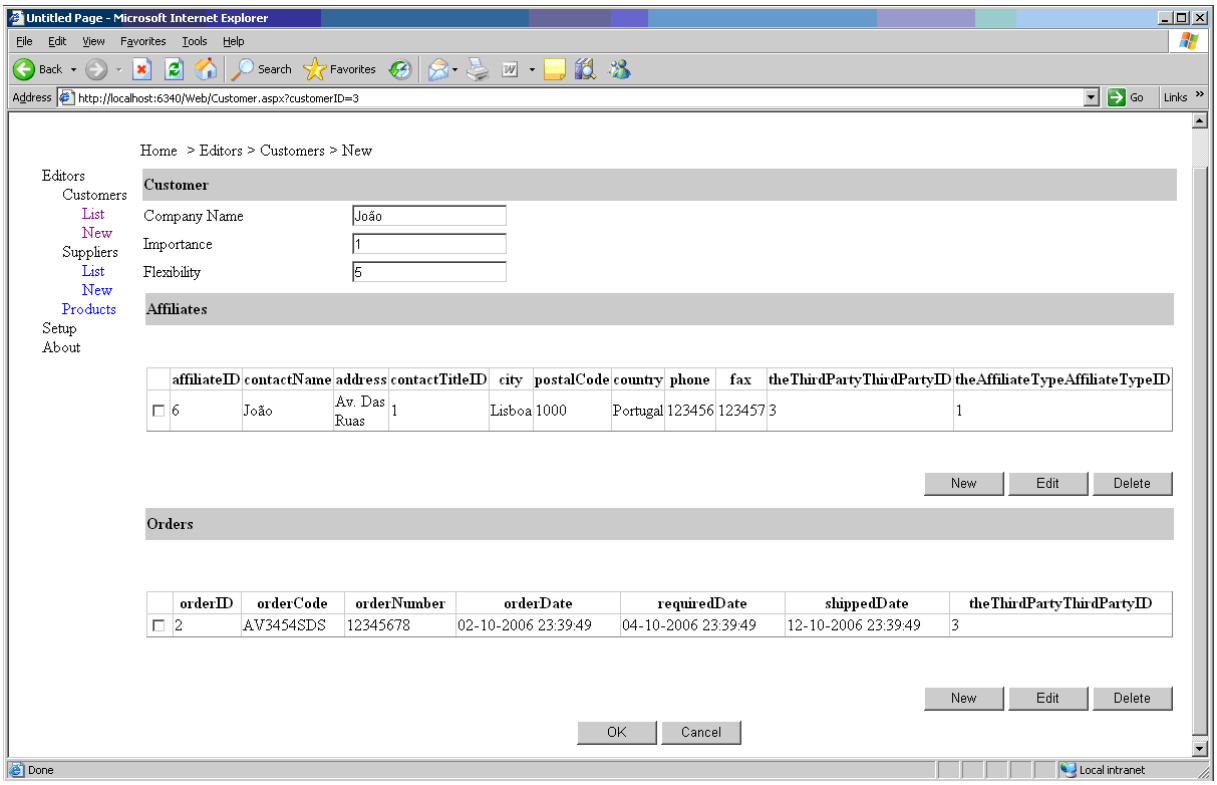

Figure A.16: Customer editing screen (ASP.NET).

Appendix B

ProjectIT-Studio Designer's Manual

Appendix C

ProjectIT-Studio Programmer's Manual