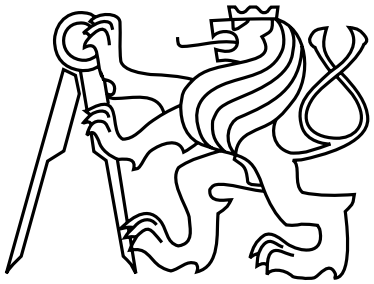
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Bachelor's Project

Netbeans plugin for class modelling

Lukáš Vyhlídka

Supervisor: Ing. Ondřej Macek

Study Programme: Software Technologies and Management

Field of Study: Computer Engineering

May 22, 2011

Acknowledgements

I would like to thank all the people who helped me with this project. At first, I would like to thank my supervisor Ing. Ondřej Macek for a lot of useful advice and hints and for the time which he has given me every time I needed. I would also like to thank Mr. Randy Schoof for the grammar correction. And last but certainly not least I would like to thank my parents for their support.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague, May 22, 2011 .....

Abstract

This Bachelor project realizes the class modeler for Netbeans platform. The plugin allows to create both platform independent and platform specific model. Plus, the plugin allows to include annotations of classes, attributes and methods into the model. Created models are provided to other Netbeans plugin through defined interface. The graphical representation of the editor was created using the JGraph framework.

Abstrakt

Bakalářská práce realizuje modelář tříd pro platformu NetBeans. Tento plugin umožňuje vytváření platformově závislých i nezávislých modelů tříd a navíc umožňuje do modelu zahrnout anotace tříd, atributů a metod. Vytvořené modely jsou poskytnuty ostatním pluginům prostřednictvím definovaného rezhraní. Grafická realizace editoru byla vytvořena s využitím frameworku JGraph.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The main emphasis of the software engineering is (in addition to the customer needs and in addition to meet the defined development time, price and the desired quality) on the reusability, maintainability and testability of the software, its rapid development, etc. One important part of the software development is the model creation (or modeling). The model depicts the most important parts of the software and helps the developer to understand the problematic. Probably the most known and the most used model is the UML[1] Class Diagram. There are a lot of tools which are able to create it but very often they are under commercial license.

This bachelor thesis deals with an implementation of Netbeans plugin which will be used for class modeling. I chose this theme because of two main reasons. The first reason is that it is not a standard information system (or something similar) that can be created with no need to learn new technologies (these systems are based on usage of well-known common frameworks). In this project I will learn to work with new frameworks like JGraph (section 4.2) and Netbeans Platform (section 4.1). The second reason is that this project can produce an useful system which can be extended e.g. within the scope of another thesis. Of course, the result of one bachelor thesis cannot produce more sophisticated system than the existing (paid) ones, but it can provide a good basis for further expansion. This project is a part of the bigger whole - there are several projects that create other modeling tools. All of these tools could be joined together in the future.

At the beginning I would like to point out that this work is not a recherche one. I didn't try to compare several possible solutions of the problem but I tried to create a solution of the problem. Despite the fact that I have been inspired by several existing systems, these systems are not described in a detail. In this thesis I focused on the description of the implementation and thus it should be easy to implement similar system according to it.

The structure of this thesis is divided into several parts. The first part brings some information about the class diagram. Next part is called *Analysis* and contains the information about the functionality of the program from the view of the user and a little info about existing solutions that inspired me. Third part is about used frameworks. Fourth part is called *Application Architecture and its Implementation* and consists of implementation doc-

[1] UML - Unified Modeling Language

umentation. In this part there is a description of how the system works inside and of some interesting subsystems. The last part is about testing.

At the end of this chapter I would like to point out that all class diagrams shown in this thesis were created by this application (called Indepmod Class Notation Plugin). I think that it is a good proof that my implementation really works.

Chapter 2

Class Diagram

A class diagram is a basic UML ([15]) diagram. It is probably the best known of all UML diagrams. The class diagram is used for description of system's (or program's) structure by showing its classes and their relations.

Despite the fact that the diagram is called class diagram, it does not describe only the classes of the system. It describes all the data types like interfaces, enumeration types, etc. Concrete types (class, interface, etc.) are distinguished by stereotypes. All these data types are in the class diagram displayed as rectangles. These rectangles can be divided into three parts:

- The first part contains the name of the class (or interface, enumeration, etc.) and eventually the stereotype (a text in «angle quotes») which describes a special property (interface, enumeration, or something else). If there is no stereotype, it means that the rectangle represents a simple class.
- The second part contains the list of attributes. Every attribute defines its scope (public, private, protected), name and the data type. There don't have to be all attributes in this part, but only the important ones.
- The last, third, part contains the list of methods. Every method defines its scope, name, list of parameters and the return value. Again, there can be only methods which are important to depict a concrete problem.

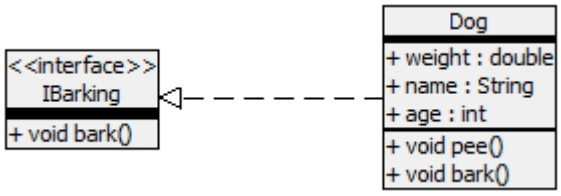


Figure 2.1: Example of a class diagram

Example of a class diagram can be seen in the Figure 2.1. There are two rectangles. First, with name `IBarking`, represents the interface (it has the «interface»stereotype). This

interface defines one public method with no return type (void) called `bark`. The second rectangle represents simple class (there is no stereotype) called `Dog`. The `Dog` class defines three public attributes (`weight` of the double type, `name` of the String type and `age` of the integer type) and two public methods (`pee` and `bark`), both with no return type. Between these two rectangles there is a relationship defining that the `Dog` class implements the `IBarking` interface.

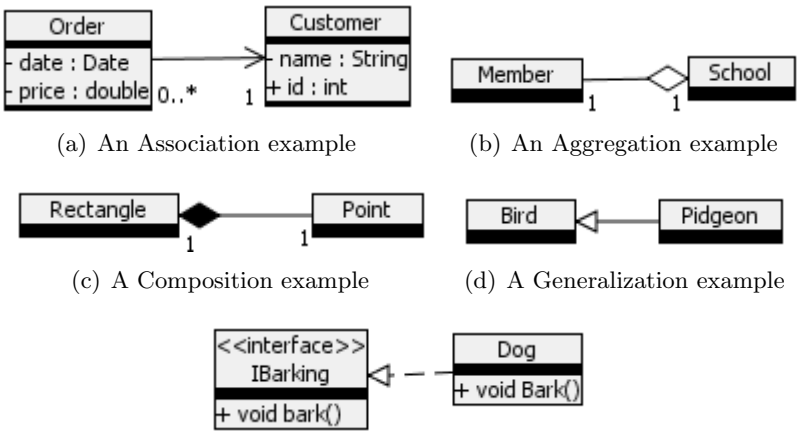
As it has been already said in the last paragraph, there can be relations between classes. UML Class diagram defines 5 basic relations between classes (or interfaces and enumerations) which define some feature of the class. These relations are:

- Association - this relation type defines a feature of the source class[1]. In the class diagram it is depicted as a solid line which leads from the source class to the target class (the target class defines the data type of the feature). The relation can be unidirectional (only source class has the pointer to the target) or bidirectional (both source and target class has the pointer to each other). The relation can also have a simple arrow defining the relation's direction (unidirectional or bidirectional). On the end of the direction (the side where the arrow is) there is the name and cardinality of the feature. You can see an association example in the Figure 2.2(a).
- Aggregation - defines that the target class is a part of the source class. This relation is depicted as a solid line with a blank diamond on the start (on the side where the starting/owning class is). An example of the Aggregation is shown in the Figure 2.2(b).
- Composition - represents stronger relation than the Aggregation. Composition tells us that when the owner object (instance of the class) is destroyed, the owned object will be destroyed as well. Composition is depicted by a solid line with a filled diamond on the start. Figure 2.2(c) shows an example of the Composition.
- Generalization - this relation defines that source class is derived from the target class. There are some restrictions for this relation. Class can be derived only from class (not from interface) and interface can be derived again only from interface. Realisation is depicted as a solid line with a solid arrow on the end. An example of this relation can be found in the figure 2.2(d)
- Realisation - defines that a class (not an interface) implements an interface. Realisation is shown as a dashed line with a solid arrow on the end. The example is shown in the Figure 2.2(e)

2.1 Meta-Model of a Class Model

The UML defines a notation and a meta-model. The notation is the graphical representation which can be seen in diagrams (e.g. in the class diagram). The meta-model is used to define the concepts of a model. It is represented by a diagram (very often it is a class diagram)

[1] In this chapter when I write the source or the target class, I will not mean only a class but also an interface or an enumeration. If it is not I will point it out.



(a) An Association example

(b) An Aggregation example

(c) A Composition example

(d) A Generalization example

(e) A Realisation example

Figure 2.2: Relations examples

which depicts all of its components. Meta-model is very important for developers of CASE[2] tools because it defines the UML structure, which can be translated into the code. Person which only creates (paints) the diagram does not need to be bothered by the existence of meta-model.

The UML's meta-model is very complex and is not shown here. It depicts the UML structure on very abstract level. Due to this fact it can be quite easily extended. For information about class model meta-model I used the information from [3] and [4]. Some information can be found also in [15].

2.2 Platform Independend and Platform Specific Model

In a computer engineering, class diagram is used for both conceptual modeling (business modeling) and class modeling on platform level, which is consequently translated into a programming language (into the programming code).

Conceptual modeling is frequently used in early stage of software/system development. Its purpose is to depict the basic ideas of the domain that we want to build (a problem domain). It is why is this model often called Domain Object Model or simply Domain Model. The Domain Model represents the vocabulary and key features of the problem domain. These features are represented by entities that occur in the problem domain, their attributes and relations between them. This type of model is really useful as a communication tool between several members of business team or between the business and technical team. By this model it can be also verified that the system meets the user's requirements, but only if this model is created well.

Class modeling on platform level describes the physical structure of the software. It uses all elements of UML class diagram, not only classes, its attributes and relations between them. The platform is meant a specific programming language like Java or C#.

[2]CASE - Computer-Aided Software Engineering

Each platform has its own features, e.g. the names of standard data types. If the class diagram is meant to be translated into the code, it has to include these platform specific features. More information about class diagram can be found in [3].

2.3 Class Modeling Tools

There are a lot of tools that provides class modeling. Some of them can be used for free, some of them not (most of the better). This section will show several of them.

The application that inspired me most was Enterprise Architect from Sparx Systems [13] company. This application does not provide only class modeling functionality. It is a CASE[3] system which is used for purposes of analysis and implementation. The Enterprise Architect provides all UML diagrams (Class diagram, Activity diagram, Use Case diagram, Sequence diagram, etc.) creation and much more. But only the Class diagram is important for this project because it is what this thesis deals with. The class modeling in Enterprise Architect has a lot of functionality that cannot be created during one bachelor thesis. Thus, it has to be decided which functionality will be implemented and which will not. The things that inspired me were e.g. the tool chooser panel or the way of class (or interface/enumeration) editing. Next list shows several tools which provides a class diagram modeling.

- Enterprise Architect - Professional CASE tool for easy creation of UML diagrams and much more. More info in [13].
- ArgoUML - Open Source UML Modeling tool based on Java programming language. More info in [2].
- MagicDraw - Business process, architecture, software and system modeling tool with teamwork support. More info in [8].
- StarUML - Open Source UML Modeling tool running on Win32 platform. More info in [14].

[3] CASE - Computer Aided Software/System Engineering

Chapter 3

Analysis

The analysis is a very important part of the whole project. It helps us to understand what is excepted from the system and how we will try to fulfil these expectations. There are many methodologies in area of software engineering but you have to do the analysis before you start to write the code no matter which methodology you choose. The analysis doesn't have to be big (especially in an agile methodology) but it has to be there.

In this chapter there is analysis of desired functionality, used frameworks and development environment. The most important part is the analysis of desired functionality because it helps to understand what is and what is not excepted from this project. The analysis of used frameworks is based on desired functionality because used framework has to be able to help with its realisation.

3.1 Desired Functionality

There are a lot of applications that provides the class modeling. Basic appearance is mostly common. There is a space where the model is shown (this space will be called workspace) and some toolbox where user can choose the tool he wants.

Every class diagram modeller has to be able to depict the basic elements of the class diagram as were described in section 2. Existing modellers provides another functionality like information about complexities of single classes, their constraints, statuses, requirements, notes and many others. It is a lot of functionality that can not be created during one bachelor project but it can be implemented in a related project.

Main aim of this project is to implement a meta-model of the class diagram. Developers of CASE system does not treat UML diagrams like diagrams. For them, the basics of UML diagram is the meta-model and the diagram is only its graphical representation. The meta-model represents the structure of the diagram and is standardly represented by an UML class diagram. The meta-model is not used only in conjunction with UML diagrams. The meta-model can be used to represent the abstraction of any other problem. The reason why I use the meta-model in conjunction with the UML diagram is that the OMG[1] defined its UML standard by use of meta-model. The structure of the meta-model that will be implemented in this project is described in next section 3.1.1.

[1] OMG - Object Management Group - is a consortium which created (among others) the UML specification.

A feature that is not a common one and is implemented in the Indepmod Class Notation Modeller is an annotation modelling. It provides the ability to annotate a class, a method or an attribute. Annotations are used in the Java programming language[2] from version 1.5 and in the C# programming language[3]. Annotations are form of metadata which are stored in the source code. This metadata specifies another features of that class/method-/attribute. Alternative to annotations are configuration files (very often of XML type) that hold these information. Annotations (or XML configuration files) are often used by frameworks. An example can be the JPA[4] framework of Java EE[5] that uses these metadata for the definition of object relation mapping (ORM)[6]. In a former version of java, these metadata were stored in XML configuration files (very often inside one XML file). When there was a lot of classes to be configured, these configuration files were very large and confusing. On the other hand, annotations allow to specify these configurations inside the classes. This means that if you want to see or edit a class configuration, you don't have to search it in a file with maybe hundreds or thousands lines. Instead, you will open a desired class and find these information right there.

Annotations are nowadays very often used. The annotation modeling feature brings the advantage that a developer can create these annotations during a class modelling. When he creates the code (lets the case tool to generate it) from the diagram he will have these annotation right there. If the case system were not able to model annotations, the developer would have to take a look at the diagram, remember which annotations should be where (in which class/method/attribute) and add them into the code manually. Of course, this should be repeated when the model is changed.

The Indepmod Class Diagram Plugin will be also able to provide its diagram data to another Netbean's plugin. It means that there could be a plugin which will be able to e.g. generate a code according to the model or create a report. This will be done by some public API[7] which will be provided by this application. This API will provide the type of the diagram (class or business diagram), the information about elements of the diagram and the picture of the diagram. The picture can be used by another plugin for example for a report. Concrete implementation of the API will be described in the Architecture chapter, concrete in the section 5.1.1.

At the end of this section I would like to recapitulate four main requirements on the Indepmod Class Diagram Plugin. These requirements are:

- The system will allow to create a class model according to the chapter 2.
- The system will allow to create both platform independent model and platform specific model.
- The system will provide an annotation modeling functionality.

[2] Annotations are in Java shown with commercial at sign, e.g. @Entity
[3] Annotations are in C# called Attributes and are shown in [square brackets]
[4] JPA - Java Persistence API
[5] Java EE - Java Enterprise Edition
[6] Object Relation Mapping - a technique that maps objects of object oriented software into tables of relation database
[7] API - Application Programming Interface

- System will provide an API which will allow other Netbeans plugins to access created diagram.

3.1.1 Class Diagram Meta-model

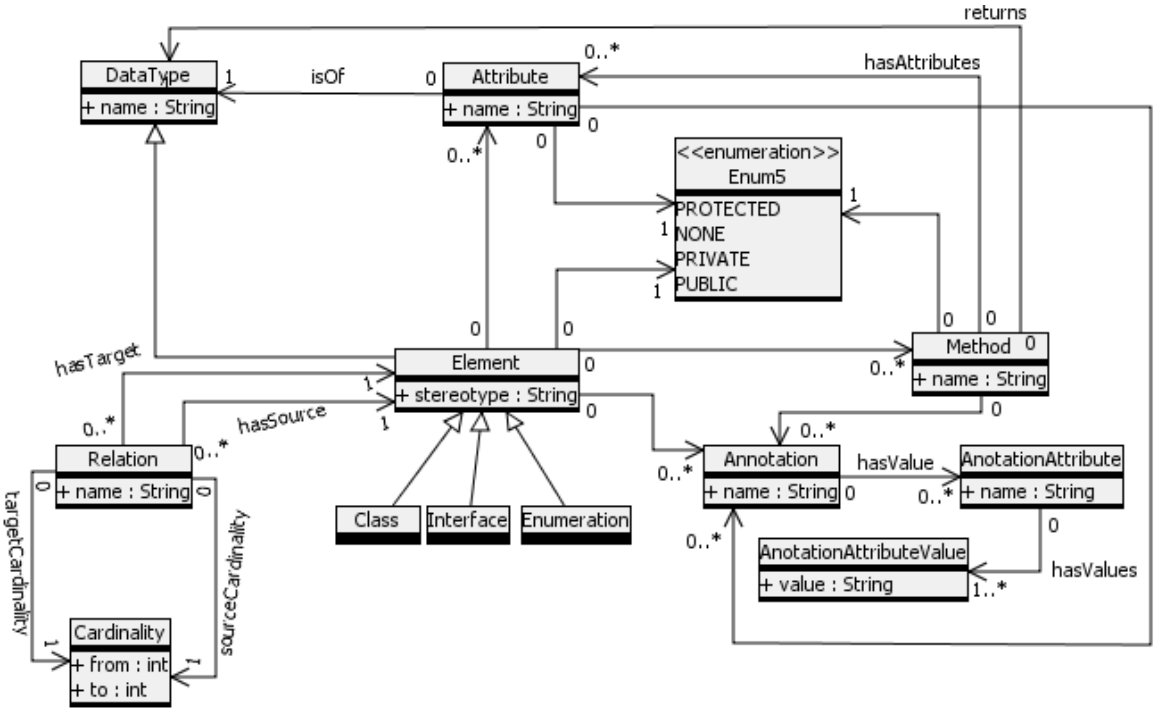


Figure 3.1: Class Diagram Meta-model

Figure 3.1 shows the meta-model of the class diagram that Indepmod Class Notation uses. From this meta-model you can see what the Indepmod Class Notation can and what it can not to model. Main building block of whole class diagram is an Element. The Element can be a Class, an Interface or an Enumeration. Every Element has a visibility, a stereotype (which can be empty) and a name. Each Element defines a data type identified by its name. In addition to the stereotype and the name, the Element has also the list of annotations, attributes and methods.

An Attribute has a name, a visibility and represents a data type. This data type can be the data type of an Element that we have already created or it can be another else data type (like String or int data type in Java). The Attribute can also have several (or none) Annotations. A Method is also defined by a name and has a visibility. The Attributes owned by the Method represents the parameters and the data type represents what the method returns. An Annotation has a name and can have several Annotation Attributes. Each Annotation Attribute has the name and a list of values.

Chapter 4

Implementation Platform and Frameworks

4.1 Netbeans API

All desktop applications have several common things. They use for example windows, menus (file, edit, etc.), docking panels, help system, online update system, etc. This functionality can be created over and over again for each new desktop application but it takes a lot of time and effort to create it. And this is the reason why frameworks are used. Advantage is that a developer (or a group of developers) creates and updates a framework that solves certain problem. Another developer can then use this framework so he does not have to implement it again. If the developer solved the problem himself, he would either create not so robust solution or spend a lot of time (maybe years) to create it. Another advantage of framework using is that it is better tested and errors are repaired in one place. If the framework is used by lot of developers, they can find another bugs, which was not found during the testing part of development and report them to the framework's creator. He can consequently repair the bug and provide another version of the framework.

The Netbeans platform is an open source application framework based on well known Swing Technology. It can be used to simplify the desktop application development by providing many techniques and design patterns. The Netbeans platform gives the developer a tool to create a robust desktop application much faster and better. Thus, the developer can focus on the application's business logic creation instead of dealing with the work that has been already done.

Basic concept of the Netbeans API is that the application can be divided into several loosely coupled modules. This is very useful especially when the application starts to be complex. Netbeans API uses a virtual filesystem (hierarchical structure of directories and files), also called central registry, for its configuration. Every module which is deployed into the application can add some information into this virtual filesystem and thus it can add some functionality. This is described later in the section 5.2.

Another useful feature of the Netbeans platform is the extended visibility settings. We can set which parts of a module will be public and which will not. This means that we can use the public visibility inside the module and specify which package will be visible

outside the module. The result of this is that we can access properties of a class directly and we don't have to worry that some other package could do the same thing.

The Netbeans platform can be used either to create the whole desktop application or to create a plugin for an existing one. Both whole applications and plugins can be consisted of several (or of many) independent modules. For this project I chose the creation of a plugin. The Netbeans platform provides a lot of functionality and if you want to know more about it, please take look at [11].

4.2 JGraph

JGraph is an opensource graph visualisation library which is based on Java Swing technology. It gives the developer great means to create a generic diagram, which is consisted of nodes and edges. JGraph provides several prefabricated components for graph node shapes or for edges shapes that connect these nodes. So if you want to create simple diagram you can use these prefabricated components. Of course, there is the possibility to create your own shapes too so if you want to create a diagram with nodes of some special shapes you can do it.

Except the diagram visualisation, JGraph provides some other functionality like zooming of the drawing area, undo/redo support, drag&drop of nodes and edges and many other. This framework will be used for visualisation of the class diagram. More information about this framework can be found in [16].

4.3 Environment

Because this thesis is about the Netbeans plugin creation and Netbeans platform is based on the Java language, there wasn't any other option than use the Java language. But it is not bad at all. Java is widely used object oriented programming language and thanks to its portability, the application can run on a lot of operating systems like Windows, Linux and so on.

As a programming enviroment I use the Netbeans IDE. The Netbeans IDE is the best choice because it is created on the same platform as the plugin. Netbeans IDE provides the full support (Wizards, etc.) for the Netbeans plugin (or module and whole applications based on this platform) creation. The Netbeans platform is described in section 4.1.

For source code version control I use Git versioning system. Git is an open source distributed version control system with emphasis on the speed and efficiency. It was created by Linus Torvalds who created it for purposes of Linux kernel development. As a remote repository I use Github[1]. For more information about Git version system take a look e.g. in [5].

[1] web pages of Github are http://github.com/

Chapter 5

Application Architecture and its Implementation

In the Figure 5.1 you can see the print screen of the running application. The whole window system (window, menus, etc.) is provided by Netbeans platform. On the left side (number 1) you can see the project tree, which is the standard component of Netbeans platform (frequently used e.g. in Netbeans IDE). Next to the project tree there is a workspace (number 2) where user can create and manage his/her class model. On the right side there is a panel (number 3), called Tool Chooser, which is used for tool selection (tool for new class addition, etc.).

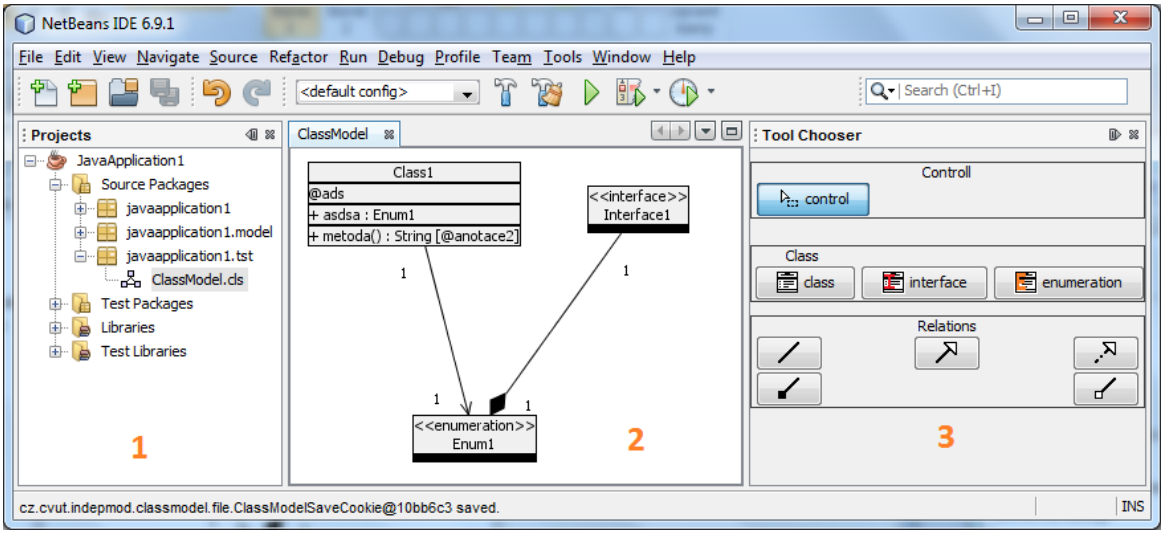


Figure 5.1: Screenshot of the program

Indepmod Class Notation plugin consists of several modules which provides its functionality. The plugin is divided into modules because some parts (e.g. API of the plugin) should be public and some parts shouldn't. Public module means that it can be used in

other plugins (or other modules of a plugin). This module division helps to comply with rules of loose coupling and high cohesion. These modules are:

- Editor - the main module of the Indepmod Class Notation plugin. This module takes care about whole class modeling. It handles the workspace and its user interface can be seen in the Figure 5.1 between Project Tree and Tool Chooser panels. The workspace has a JGraph inside which controls whole class modeling.
- API - this module provides a public API (interfaces). Implementations of these interfaces can be found (e.g. by other Netbeans plugins) in the Lookup of the workspace (part of the Editor module). This module is public so it can be used (imported as a library) by any other plugin.
- Tool Chooser - allows user to choose a tool of actual selected workspace. The user interface of the module is a panel with single tools in it (e.g. a class or a relation between classes).
- JGraph - this module is used as a library wrap for JGraph (it allows other modules to use JGraph framework).

Figure 5.2 illustrates the relation between Editor and ToolChooser modules. Editor provides instances of `ToolChooserModel` and `IClassModelModel`. These interfaces can be used by other plugins/modules. One of these modules is ToolChooser which uses `ToolChooserModel` to set the desired tool of active workspace (Editor module). These interfaces will be described in detail in following section.

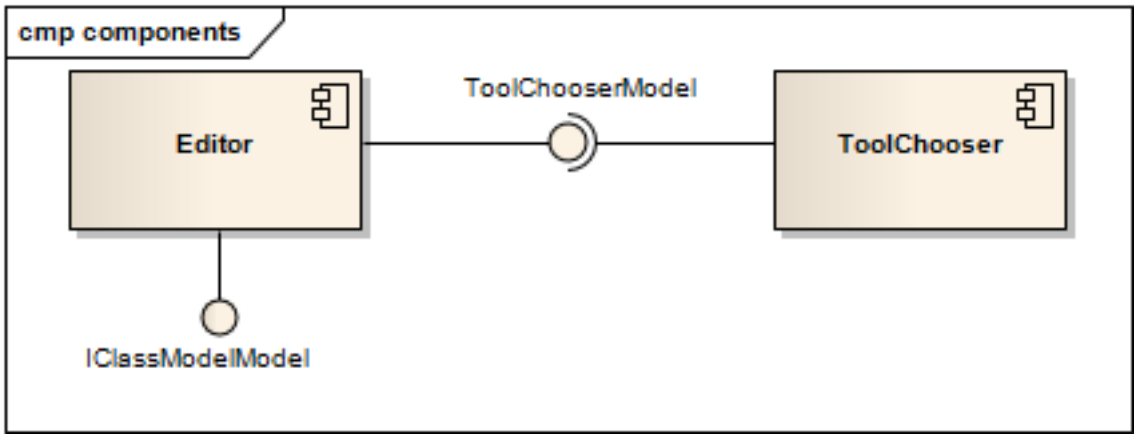


Figure 5.2: Component diagram of Editor and ToolChooser Modules (created in Enterprise Architect)

5.1 API Module

This module defines interfaces whose implementations can be found in the lookup of the workspace (workspace is the part of the editor module). Purpose of these interfaces are

described in next sections. This module is public so it can be used by any other module (or any other plugin). There are two packages in this module:

- cz.cvut.indepmod.classmodel.api.model
- cz.cvut.indepmod.classmodel.api.toolchooser

5.1.1 ClassModel API

This package contains interfaces whose implementation can be used to access the information about class model. Lookup of the workspace contains the implementation of the `IClassModelModel` interface. This object provides the information about the type of a model (business model / class model), list of elements from the model (`IElement` interface) and the picture of the model. `IElement` represents a class, an interface or an enumeration. `IElement` provides the list of annotations (`IAnotation` interface), attributes (`IAttribute` interface), methods (`IMethod` interface) and relations to other classes(`IRelation` interface). And so on with the attributes, etc. These interfaces represents the structure of the class model meta-model, described in section 2.1.

Relation between (two) classes is represented by instance of `IRelation` interface. `IElement` returns the list of `IRelations` that belongs to it. `IRelation` holds information about its type (association, composition, agregation, generalization, implementation) and about what element is at the beginning and at the end of the relation, including cardinalities.

Cardinalities are represented by instance of `ICardinality` interface. The `ICardinality` interface has two methods (`getFrom()` and `getTo()`) which return from and to value (both are of integer type). Sign of infinity (e.g. in the 1..* or * cardinality) is treated as -1. In the Table 5.1 you can see examples of return values of some cardinalities.

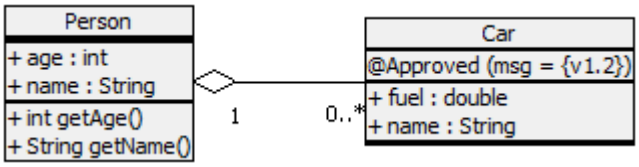
Cardinality	getFrom() result	getTo() result
0 (or 0..0)	0	0
1 (or 1..1)	1	1
4 (or 4..4)	4	4
* (or 0..*)	0	-1
1..*	1	-1
0..1	0	1
2..5	2	5

Table 5.1: Examples of cardinalities

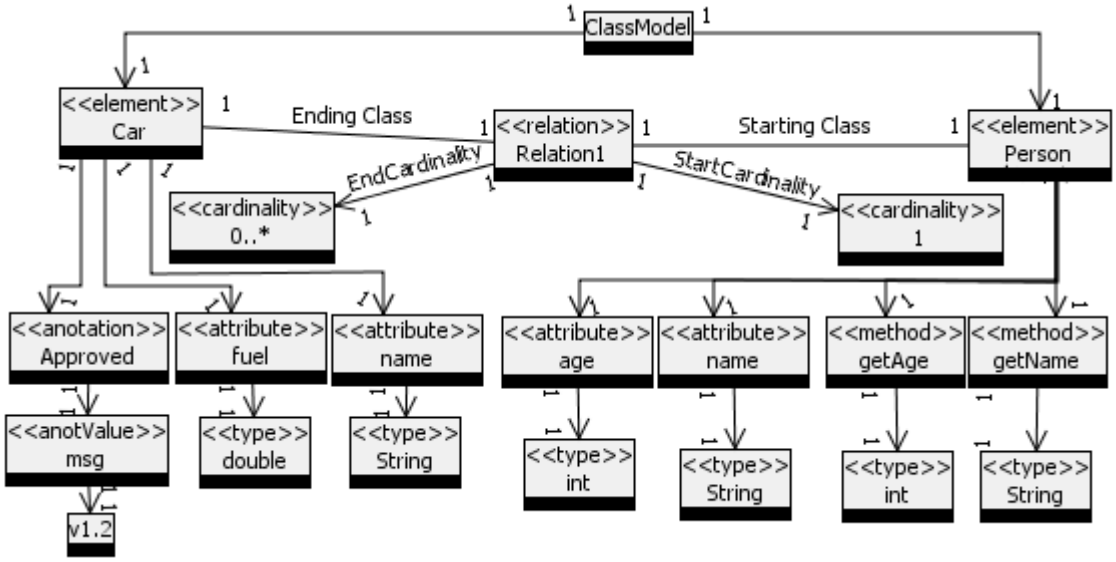
The ClassModel API can be treated as a tree. The root node of the tree is `IClassModelModel` instance whose children are elements, etc. You can see an example of a class diagram in the Figure 5.3(a) and its tree representation in the Figure 5.3(b).

5.1.2 ToolChooser API

ToolChooser API is used to hold information about the selected tool of Editor module. The main part of this module is `ToolChooserModel` class. Instance of this class represents the



(a) A class diagram



(b) A tree representing the class diagram

Figure 5.3: Example of ClassModel API Tree

actual selected tool. `ToolChooserModel` implements the Observer design pattern so you can register your listener which will be called anytime the model changes.

The API is used by both ToolChooser and Editor modules. Workspace (part of Editor Module) has an instance of `ToolChooserModel` in it's lookup. This means that Editor module leaves the tool selection on someone else.

ToolChooser uses this instance (from the lookup of active workspace) to set the desired tool (e.g. new class). But it is also possible to make another plugin that will do this (e.g. in the toolbar). Structure of this package is really simple and you can see it in the Figure 5.4.

5.2 ToolChooser Module

This module is used for setting of demanded tool of active workspace. The user interface of this module can be seen in the screen shot of the running application (Figure 5.1).

ToolChooser searches for instance of the `ToolChooserModel` in the lookup of active workspace (Editor module). When the instance is found, ToolChooser can set its tool (new

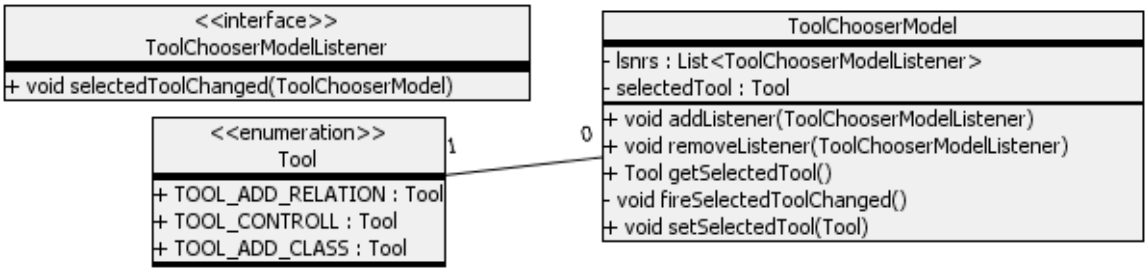


Figure 5.4: Class Diagram of ToolChooser API package

class, new relation, etc.). This means that it can be used to set the desired tool of that workspace. The structure of this module can be seen in the Figure 5.5.

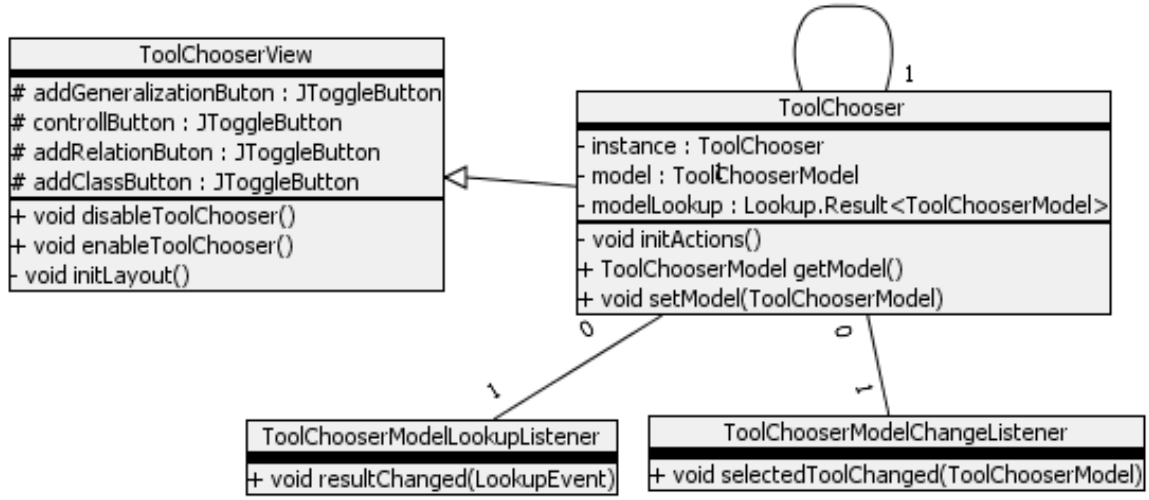


Figure 5.5: Class Diagram of ToolChooser Module

`ToolChooser` is a singleton which is inherited from the `TopComponent` class. `TopComponent` is provided by Netbeans platform. It is derived from standard `JComponent` class (javax.swing) and adds many methods which can be used for controlling a window. Ancestors of `TopComponent` class can be used as a user interface (component that is visible to the user) in the Netbeans platform application. More info can be found in [11] in chapter called Window System).

In the layer.xml file there is set the action for opening the ToolChooser (in running application can be accessed in menu Windows → Tool Chooser). Layer.xml is a configuration file of the Netbeans platform, which is provided by modules. Content of this file defines new folders and files which will be inserted into the system filesystem (a central registry) when the module is loaded. The system filesystem is a virtual filesystem, which is used for Netbeans platform settings. For example there are some folders that are used for Window System API (window positions, etc.) or Actions API (e.g. actions in the menu) configuration. More info

about this can be found in [11] in the chapter 3, called Window System.

5.3 Editor Module

This is the fundamental module whose purpose is to create a workspace which provides the class modeling. For graphical part of class modeling is used framework JGraph ([6]). More info about JGraph framework can be found in [16].

5.3.1 Module Structure

Main class (part of the cz.cvut.indepmod.classmodel.workspace package), which represents the workspace, is the `ClassModelWorkspace`. Thus, if you want to study the code, you should start right here. `ClassModelWorkspace` is extended from the `TopComponent`. This class is responsible for whole initialization. It creates:

- `ClassModelGraph` - this class is extended from `JGraph` and represents the class model graph to the user. Instance of this class is situated inside the `ClassModelWorkspace` component (`JGraph` is also derived from `JComponent`)
- `ClassModelModel` - Implementation of `IClassModelModel` which returns the list of classes that are in the class model and the type of the diagram (class or business model). The purpose of `IClassModelModel` interface is described in the section 5.1.1.
- `ClassModelMarqueeHandler` - this class is responsible for user input handling which is made inside the `ClassModelGraph`.

`ClassModelWorkspace` is simply `JComponent` (`TopComponent`) which has the `JGraph` inside. `JGraph` presents the class model to the user. There are cells (representing classes) which are related together by edges (representing relations).

In the next section I will try to describe the implementation of the Editor Module. I Will start from the initialization and after that I will try to explain every part that is created during the initialization.

5.3.2 Editor Module Initialisation

As I have already said before, the `ClassModelWorkspace` class does the initialization of the whole module. There are two cases of initialization. The first case, when user creates new class diagram, and the second, when user opens an existing model from file. Both cases are very similar. They differ in the way of `DiagramDataModel` initialization. For this purpose there are two constructor variants.

The first, non parametric, constructor is used when the new class diagram (with no associated file) is created. This constructor calls the `DiagramModelFactory` which creates new instance of `DiagramDataModel` and returns it.

The second constructor is used when the user opens a file with a class model. It accepts an instance of `ClassModelXMLDataObject` (one parameter). This object represents

the file in which is the class model saved (more in the chapter 5.3.8). The constructor gets the input stream of that file and asks the `ClassModelXMLCoder` to decode its content. `ClassModelXMLCoder` decodes it and returns `DiagramDataModel` instance filled according to that file's content. `ClassModelXMLCoder` is part of the persistence layer and will be discussed in the chapter 5.3.7.

`DiagramDataModel` class represents the data which has to be persisted when user want to save his class diagram. At present, instance of this class holds the `GraphLayoutCache` instance (class of JGraph framework which holds the information about cells in the graph), list of static data types (list of default data types, like e.g. String or int, for certain language which is chosen during new class diagram file creation), list of dynamic data types (data types that user created by hand), list of stereotypes and the type of the diagram (class or business model, it is also chosen during new file creation). The `DiagramDataModel` is also used inside the Editor module, especially by forms. To fulfil the rule of loose coupling, forms does not hold the pointer to it but they gain the pointer through the lookup of active diagram. The lookup will be described later in the section 5.3.9.

After the `DiagramDataModel` is gained (created or loaded) the initialization is the same. What the `ClassModelWorkspace` creates have been already written up in the chapter 5.3.1. For better understanding you can see the sequence diagram in the Figure 5.6 which shows the initialisation of new `ClassModelWorkspace`. For simplicity there is only what `ClassModelWorkspace` creates. The processes inside these classes are not shown (Sequence diagram would have been really big) but the process of creation will be discussed later.

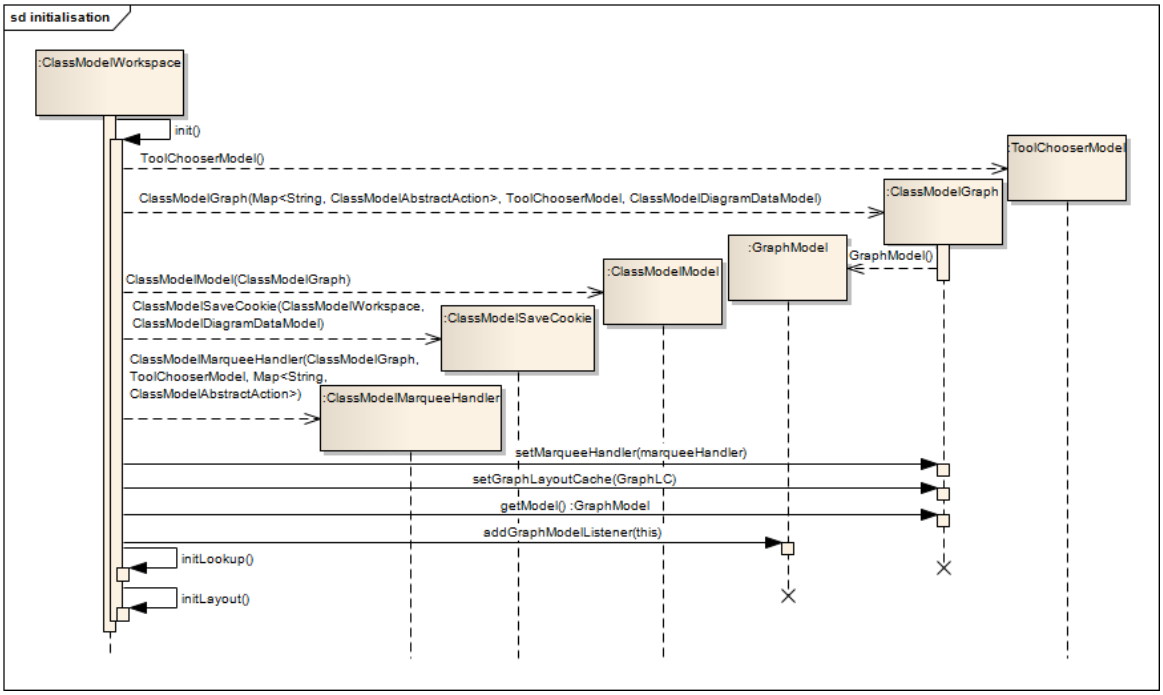


Figure 5.6: Simplified sequence diagram of ClassModelWorkspace initialization

5.3.3 ClassModelGraph implementation

`ClassModelGraph` is extended from `JGraph`, which provides a lot of useful functionality (more on this can be found in [16]). `ClasModelGraph` adds some functionality for class manipulation. This added functionality is shown in following list.

- `insertCell(Point p)` - creates new cell on desired position according to the selected tool (in `ToolChooserModel`). Usage of this function is shown in the section 5.3.5 in Figure 5.8.
- `getAllClasses()` - returns the list of classes that are in the diagram. Usage of this function will be shown in the section 5.3.5

To handle events in the graph (in `ClassModelGraph`) there is the `ClassModelMarqueeHandler`. Instance of this class is created in the `ClassModelWorkspace` and is added into the `ClassModelGraph` instance. Purpose of this class is to handle all user inputs in the `ClassModelGraph` (in the canvas of the graph) and control the popup menu (`JPopupMenu` and its content). When user does an action (e.g. click with mouse), the `ClassModelMarqueeHandler` will find out if it is a control click (e.g. cell selection), new cell addition, edge (line between cells) addition and so on. This class is also responsible for rendering of the temporary line when user creates new edge.

5.3.4 JGraph class cell rendering

In this section I will explain how I render the classes (the rectangles with the class name and lists of annotations, attributes and methods) into the JGraph workspace. But first a little theory remind. JGraph uses MVC pattern for cell rendering. Cells (the model of cell) in JGraph are represented by `DefaultGraphCell` (or its subclasses). `DefaultGraphCell` implementation stores the user object and an attribute map. The user object is an object which is associated with the cell. The Cell uses its method `toString` to render the name of the cell, so in most cases this object is an instance of String. But it can be any other object which can store another information about the cell. The attribute map is used when you want to set the appearance of the cell.

All graph cell has an associated view (`VertexView` implementation). This `VertexView` implementation associates the renderer, editor and cell handle[1] together for a cell (`DefaultGraphCell` implementation). The cell and the view is associated together by `CellViewFactory` which returns a cell view for a particular cell. More on this theme can be found in [16].

JGraph basically supports rendering of basic shapes like rectangles, ovals and so on. When you want to render your own shape, you have to create your own cell view (`VertexView`), renderer (`CellViewRenderer`) and `CellViewFactory` implementation.

So this is what I did. I created `ClassModelCellViewFactory` which is derived from JGraph's `DefaultCellViewFactory` and overrides it's method `createVertexView(Object o)`. If the object in the parameter is instance of `ClassModelClassCell` class, the method

[1] Cell handle is what appears when the cell is selected (e.g mouse click) and what allows to resize it.

returns new instance of `ClassModelVertexView` (derived from JGraph's `VertexView`). Otherwise the method returns the default view (calls its parent's method). `ClassModelVertexView` returns instance of `ClassModelVertexRenderer` which is derived from `VertexRenderer`. `ClassModelVertexRenderer` overrides its method `getRendererComponent`. This method returns new instance of `ClassComponent`. `ClassComponent` is extended from `JComponent` and renders the class according to the `ClassModel` user object.

These classes can be found in the package cz.cvut.indepmod.classmodel.workspace.cell. Instance of `ClassModelCellViewFactory` is inserted into the `GraphLayoutCache` during the workspace initialisation. You can see the class diagram in the Figure 5.7.

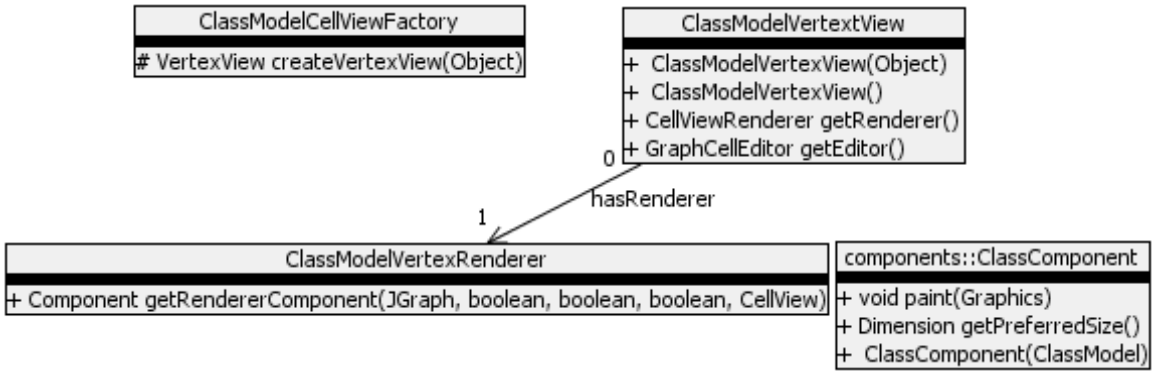


Figure 5.7: Structure of classes which render the class cell

5.3.5 ClassModel API implementation

Implementation classes of ClassModel API (its interfaces are in the API Module) are situated in the cz.cvut.indepmod.classmodel.cell.model.classmodel package. Main problem I had to deal with was to design where to store the data of the ClassModel.

Basically, `ClassModel` class is used as a User Object for JGraph cells. User Object (`ClassModel` instance) does not have normally the pointer to its cell (only cell has the pointer to it's user object). `ClassModel` holds information like name of the class and list of its attributes, methods and annotations. But where to store the information about relations with other cells (classes)? This information is stored in the JGraph (in its model). So the first idea was to copy this information into the `ClassModel` instance when a relation is created. But this is not very nice because of data duplication. The second purpose was to add a pointer to cell into the `ClassModel` instance. But how? The answer is quite simple. I created the `ClassModelClassCell` which extends `DefaultGraphCell` of JGraph. The extension of this class is that it adds an pointer to itself into the User Object if this User Object is instance of `ClassModel`.

So problem is solved. Some information like the name is stored inside the User object and some like the relations are gained from the JGraph cell. In the Figure 5.8 you can see how is created a class when user selects new class tool in the ToolChooser and clicks somewhere in the Editor workspace.

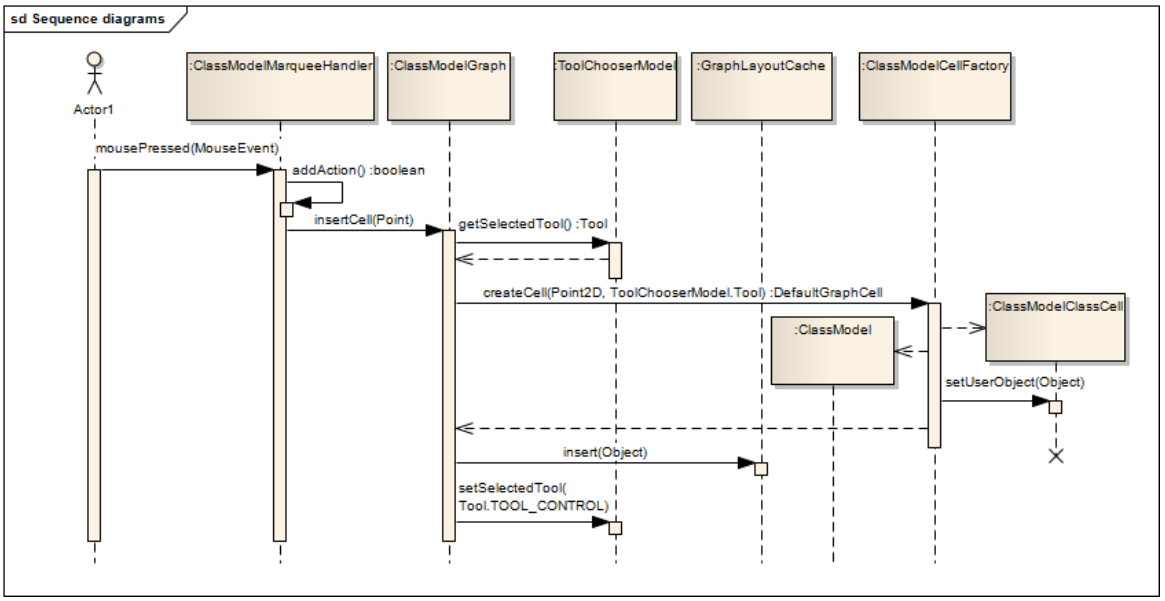


Figure 5.8: Sequence Diagram of class creation

5.3.6 Dialogs

Dialogs of the Indepmod Class Notation plugin is created on Swing technology. Their classes are located in cz.cvut.indepmod.classmodel.frames.dialogs package.

Every dialog uses the special case of MVC[2] design pattern. Standard MVC design pattern has three single components:

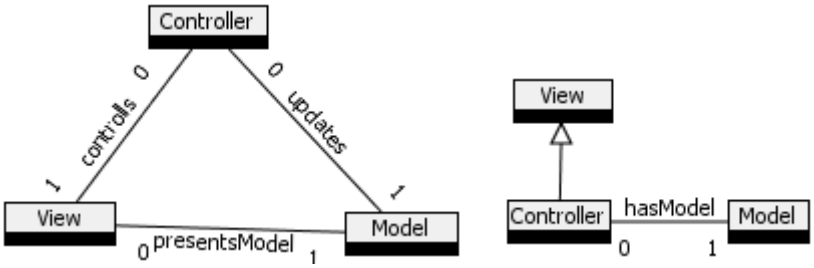
- a model - represents the data that is processed. It can be e.g. a class instance that holds this data.
- a view - presents the data to the user. It is e.g. a form with single components but it does not have to be only the form. It can be also an image or a chart. The view holds the pointer to its model so it can be refreshed when there is a change in the model.
- a controller - has a pointer to both the view and the model. It performs an action when user does something (e.g. click on a button in a view) and updates the values in the model.

The MVC pattern in this project is slightly different. The view and the controller are not separated but the view is a predecessor of the controller. It is done so because it is not expected to change the view for another else. This does not violate the rule of encapsulation and in addition, the form elements (text fields, buttons, etc.) can be in the view created with protected visibility. It means that the controller can manage these elements directly. You might be asking why is used this amended version of MVC pattern? Answer is, that this project was not created upon the Netbeans API from the beginning. At the beginning,

[2]MVC - Model View Controller

this project was a plugin for a Promod application[3] and this type of MVC pattern were used.

For better view of the MVC design pattern you can take a look at the Figures 5.9(a) and 5.9(b) where is shown the principle of this design pattern. Detailed info about this design pattern can be found in [10].



(a) Standard MVC Design Pattern

(b) MVC Design Pattern used in the Indepmod Class Notation

Figure 5.9: Class diagrams of MVC Design patterns

Another design pattern that some forms use is the Abstract Factory in conjunction with Template method. The Abstract Factory design pattern solves the problem when we want to create an implementation A of an interface under any conditions and another implementation B of the same interface under any other conditions. It consists of three parts. The first part is an abstract class which is the abstract factory. This class defines the interface for concrete classes (concrete factories) and a static method which returns an implementation of concrete factory. This static method can accept some parameters from which it decides which concrete factory to return. The second part is consisted of these concrete factories. Their responsibility is to create single components according to the interface of the abstract factory. The third part is consisted of components which are created by these concrete factories.

You can see an example of Abstract factory design pattern in the Figure 5.10. Imagine that your application will be used on both Windows and Linux and that you want to use different GUI component on these platforms. The solution is depicted in the example. You will create GUI components by use of `AbstractGUIFactory`. `AbstractGUIFactory` will find out on which platform the application is running and returns either `WindowsGUIFactory` or `LinuxGUIFactory`. Nevertheless, you won't think much about what factory implementation is returned. These concrete factories are returned as their parent class type - AbstractGUIFactory. The only thing you will do is call appropriate method on the returned object which will get you the component you wish. And that is all. For better imagination I will show an example in the Appendix B.

Usage of this design pattern was necessary because Indepmod Class Notation creates two types of diagrams - class and business diagram. The class diagram lets the user edit all elements of classes (attributes, annotations, methods) but the business diagram lets the

[3]Promod is the application of the master's thesis written by Petr Zvěřina.

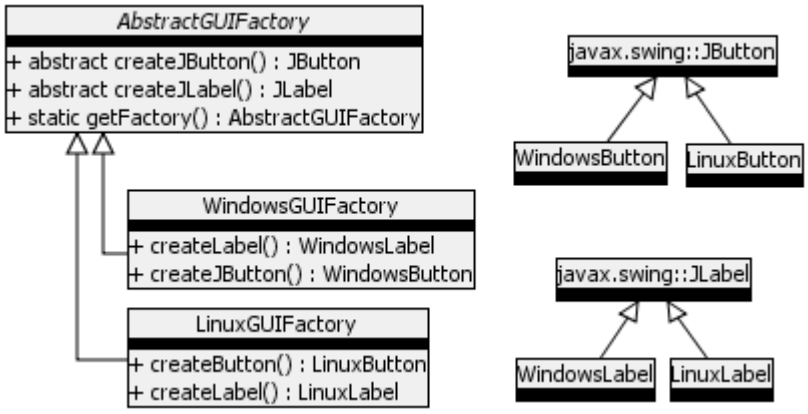


Figure 5.10: Abstract Factory design pattern example

user edit only the attributes. When user make the action to edit the class, Indepmod Class Notation shows appropriate dialog according to the type of the model.

Forms that are based on this design pattern are forms for class edition and attribute creation. In this thesis I will explain only the principle of the form for class edition. The form for attribute creation is analogous.

The form for class edition of the business diagram is defined by `BusinessModelEditClassDialog` class and the form for class edition of the class diagram is defined by `ClassModelEditClassDialog`. These dialogs have many things in common. This is why they have a common predecessors. These predecessors looks like any other dialog except that they are abstract[4]. These abstract predecessors are `AbstractEditClassDialogView` (a view) and `AbstractEditClassDialog` (a controller).

The abstract view is able to create the whole UI for class diagram class edition. Component initialization is divided into several protected methods. These methods creates single parts of the UI. Their names are:

- `initClassNamePanel` - creates the panel for name and stereotype edition
- `initAnotationPanel` - creates the panel for annotation edition
- `initAttributePanel` - creates the panel for attribute edition
- `initMethodPanel` - creates the panel for method edition

These methods are treated as template methods so they can be overrode. The abstract controller behaves like any other controller of this program.

The `BusinessModelEditClassDialog` is extended from the abstract controller and overrides methods that creates annotation and method panels. It overrides them in a way that it returns an null pointer. The abstract view compose its UI of these parts and when a

[4]Abstract class - it can't create its instances

part is null, it is not added there. The `ClassModelEditClassDialog` does not override any method because all desired functionality is done in its predecessors.

These dialogs are created by an Abstract Factory. The Abstract Factory class is placed in the cz.cvut.indepmod.classmodel.frames.dialogs.factory package and is called `AbstractDialogFactory`. It has a static method which accepts the type of the diagram and returns an instance of a concrete factory. This concrete factory creates the dialog. Please take a look at the javadoc or into the source code of this project for more information about the names of methods, etc. For more information about the Abstract Factory design pattern take a look at the [10].

5.3.7 Persistence

Persistence layer implementation is situated in the cz.cvut.indepmod.classmodel.persistence.xml package. There is a `ClassModelXMLCoder` class that is created as a singleton. Instance of this class is responsible for encoding and decoding the `ClassModelDiagramDataModel` instance into or from a stream (`InputStream` or `OutputStream`). Its method for encoding, `encode(ClassModelDiagramDataModel model, OutputStream stream)`, is called by SaveCookie (discussed in the section 5.3.9).

`ClassModelXMLCoder` uses `java.beans.XMLEncoder` and `java.beans.XMLDecoder`. This creates the xml file that represents the steps to create the exactly same object that was encoded. Because `XMLEncoder` can encode in default only some types of objects, there are some persistence delegates that allow to encode particular object I use. These persistence delegates can be found in the cz.cvut.indepmod.classmodel.persistence.xml.delegate package.

If you want to know more about how `XMLEncoder` or `XMLDecoder` works, please take a look at [9].

5.3.8 Netbeans File Association

When you want to open a file that is associated with your application (or with your plugin) in a standard Netbeans way, you have to tell the Netbeans which file type is of you application/plugin.

Netbeans platform allows its plugins to recognize new types of files. So when you want to associate new file type with your plugin, you can do it easily. The configuration of the file association is, like any other configuration, done via the layer.xml configuration file. If you are not familiar with this, you can find it in [11] in the chapter called Data System.

The file type association settings, as have been already written, can be found in layer.xml file. At present there is settings of file type association and settings of wizard that can be used to create new class diagram file. In this wizard user sets the name of the file, type of the diagram and the language (Java, C#, etc.). Files that are associated with this plugin are XML files with .cls suffix. Content of these files will be discussed later.

5.3.9 Lookup

Lookup is (in Netbeans platform) a technique to join more plugins together with loose coupling. On the module level, lookups enable to register a service by its provider and

find the service by its consumer in a central repository. On the level of single components (`TopComponent`), lookups enable to exchange data between this component (provider) and consumers. The main difference between lookup on the module level (service registry) and lookup on the component level is that component does not provide its data by a central registry but by its own lookup object. More reading about this can be found in [11] in the chapter called Lookup.

Editor Module uses the Lookup on the component level. `ClassModelWorkspace` creates its Lookup during the initialization. It adds there an instance of `ToolChooserModel` and an instance of `ClassModelModel`.

`ToolChooserModel` instance is used by ToolChooser Module to set the desired tool. Thanks to this, there can be one ToolChooser component which manages all `ClassModel-Workspaces`. Of cource, `ToolChooserModel` instance in the lookup can be used by other plugins also.

`ClassModelModel` instance implements the `IClassModelModel` interface and can be also used by other plugins. It provides the Class Model API to the outside world. Thanks to this there can be for example another plugin that will generate the code from the class model of this module.

When user does any change in the model (add new class, new relation between classes, change the attribute of the class or something similar), `ClassModelWorkspace` inserts a `SaveCookie` instance into its Lookup. This tells the Netbeans platform that this component is changed and can be saved (save action in File menu is enabled). When user calls this action, the `SaveCookie` tells the `ClassModelXMLCoder` to save the `GraphLayoutCache` into the associated file. When it is done, this `SaveCookie` implementation is removed from the Lookup, so the Netbeans platform knows that the component does not have to be saved.

5.3.9.1 Lookup example

In this section there is an example of how to obtain the `IClassModelModel` implementation. Because Editor module uses the lookup on the component level, the `IClassModelModel` implementation can be accessed only from an active workspace. The example is an action that can be configured (in the layer.xml file) to be placed e.g. in the menu or in the toolbox.

```
1  package somepackage;
2
3  import cz.cvut.indepmod.classmodel.api.model.IElement;
4  import cz.cvut.indepmod.classmodel.api.model.IClassModelModel;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import java.util.Collection;
8  import java.util.logging.Logger;
9  import org.openide.util.Utilities;
10
11 public final class MyAction implements ActionListener {
12
13     @Override
14     public void actionPerformed(ActionEvent e) {
```

```
15          IClassModelModel model;
16          model = Utilities.actionsGlobalContext().lookup(
                IClassModelModel.class);
17
18          if (model == null) {
19              // There is no class model, inform the user
20              // that he does not have opened the workspace
21              // with a class diagram
22          } else {
23              // You have a class model, start to do something...
24          }
25      }
26  }
```

On the line 16, there is a command, that loads the implementation of `IClassModelModel` from an active workspace. `Utilities.actionsGlobalContext()` returns the Lookup of the active workspace. Active workspace is the `TopComponent` which the user is working with. This Lookup instance is consequently asked to return the implementation of the `IClassModelModel` interface.

Chapter 6

Testing

Software testing is very important part of whole development process. There has to be the quality verification in every project which is intended to be successful one. Developers has to test the application through all phases of application development. Testing has a lot of advantages. The main purpose of test is to find a bug during a development (by development I mean all phases like analysis, design, testing, etc.) or verify that all functions works as they should.

Testing could be done manually or automatically. Manual testing is suitable in case that we don't want to repeat testing procedure a lot and thus the creation of automatic test would be harder than manual testing. Automatic tests are very helpful when we want to run them repeatedly. On the other hand, the developer who would test the same functionality again and again could miss out something important.

Automatic tests can be run whenever we want and can verify that all functions works properly. Automated test are very often used e.g. when there is a new functionality added into the application. In this case we want to verify that the new functionality did not violate any of yet existing ones. Another example can be a refactoring. Refactoring is the change of the program structure without the change of it's behavior. When we run the test before (test has to work fine before the refactoring) and after the refactoring, we can find out if the refactoring was successful (all functionality still work) or not. Of course, tests will help us only if they are created properly. It means that they test each functionality in a detail.

6.1 JUnit

For automatic test purposes I use the JUnit framework. The JUnit is well known framework which is used for unit tests creation in Java platform.

An Unit test is a test which tests the units[1] of our application apart the others. In procedural programming this unit can be an function or a procedure. In Object Oriented programming (OOP), this unit is often a class. The Unit test then tests methods of this class. Despite of the fact that Unit tests are primarily intended to test units, they are very

[1] Unit is the smallest part of our application which can be tested alone. It means that it can be built and run.

often used for testing of class ensemble as well. If you don't know the JUnit framework, you can find some useful information in [7].

The question is which parts of the application to test and which not to test. The best and simplest answer is to test everything. This approach would result in 100% tested application but for what price? Tests would cover all functionality, which is great. But even the functionality which does not have to be tested, like implementation of user interface (UI) and so on. This means that the test creation would take a very long time and it is quite worthless.

In this project there are unit tests which covers only the business functionality of the Indepmod Class Notation plugin. These unit tests test single class functionality and on the other hand the functionality of class ensembles. These tests can be run at any time and can verify that all parts of the application works as they should. The reason why I don't test everything is that some parts of the application are not so dangerous and to test them would take a long time.

6.2 User Interface

Among the parts that are not so dangerous belongs e.g. the User Interface (UI). An error in UI should not affect any other part of the application. Moreover, such an error would be very quickly discovered and on the other hand to test this functionality in JUnit framework would take a very long time.

For UI, more precisely GUI[2], testing there are different tools than JUnit. Some of them are suitable for desktop application GUI testing (e.g. Abbot - more info can be found in [1]), some of them are suitable for a web application GUI testing (e.g. Selenium - more info can be found in [12]), etc.

Because I don't want to test the UI repeatedly (or not very often) I decided to test the GUI manually. Single test cases are shown in the Appendix C and their results are shown in this section in the Table 6.1. I chose two environments on which I will test the application GUI:

- Acer Aspire 3810TZ with Windows 7 Professional (64bit)
- Acer Aspire 3810TZ with Ubuntu 10.10 Maverick Meerkat (32bit)

During the development there were found some errors but they were repaired consequently. Final tests did not find any other failure. The Test Cases results are shown in next Table 6.1.

[2]GUI - Graphic User Interface

Test Case Name	Windows	Ubuntu
TC0	Passed	Passed
TC1	Passed	Passed
TC2	Passed	Passed
TC3	Passed	Passed
TC4	Passed	Passed
TC5	Passed	Passed
TC6	Passed	Passed
TC7	Passed	Passed
TC8	Passed	Passed
TC9	Passed	Passed
TC10	Passed	Passed
TC11	Passed	Passed
TC12	Passed	Passed
TC13	Passed	Passed
TC14	Passed	Passed
TC15	Passed	Passed
TC16	Passed	Passed
TC17	Passed	Passed
TC18	Passed	Passed
TC19	Passed	Passed
TC20	Passed	Passed
TC21	Passed	Passed
TC22	Passed	Passed
TC23	Passed	Passed
TC24	Passed	Passed
TC25	Passed	Passed
TC26	Passed	Passed
TC27	Passed	Passed
TC28	Passed	Passed
TC29	Passed	Passed
TC30	Passed	Passed
TC31	Passed	Passed

Table 6.1: User Interface Test Cases Results

Chapter 7

Conclusion

7.1 Future works

Indepmod Class Notation plugin is already an useful class modeling tool. It provides standard class modeling functionality, plus the annotation modeling feature. On the other hand, there are some other things that could be done on this project. I will mention some of them, but it will not be all - the imagination has no limits. The things that I will mention could be implemented in a related bachelor, master or semestral project. These new functionalities are e.g.:

- Ability to keep more information about elements (classes, interfaces, ...) and relations like their description, version, complexity, OCL[1] support, and so on.
- The list of languages (that are displayed during the diagram creation) could be loaded from an XML (or any other) file. This could be done e.g. by usage of JAXB[2] technology. This functionality would allow to simply add new language definitions (only by editing the XML file).
- User Interface (design and control) improvements. E.g. the key shortcuts addition or transformation of the tool chooser panel into the Netbeans palette.
- More options of persistence (e.g. a relational database) - possibility to choose if diagrams will be stored into a xml file, into a relational database or into something else - and optimization of existing solution (e.g. create own structure of elements to be saved).
- Indepmod class notation plugin could allow to create a hierarchic structure of class diagrams. Every level would depict the class diagram of a package. This would be very useful in conjunction with source code generation.

Because this project is a part of several related projects, there is another thing that could be done. All of these projects are aimed on the Netbeans Plugin creation. These

[1] OCL - Object Constraint Language
[2] JAXB - Java Api for Xml Binding

plugins provides, like this one, the creation of an UML diagram but they are not joined together. Some other project could solve this problem. It could be done e.g. by some root plugin. The root plugin would implement the common functionality for all notations and provide services for other plugins. These other plugins would implement the functionality for particular diagram.

7.2 Summary

In this project I dealt with the creation of the rich client class modeling tool, called Indepmod Class Notation Plugin. Thanks to this project I gain an experience with new technologies like Netbeans platform and JGraph framework for which I am very thankful.

The task of this project was accomplished properly. It means that all requirements like the possibility to create both platform specific and platform independent model, annotation modeling functionality and a public API which allows other Netbeans plugins to access created diagram were implemented. Plus, there was added another functionality like language selection in new diagram creation. This feature is very important for class diagram modeling on platform level. It has been already described in the section 2.2.

Among the project's benefit I would like to mention especially the annotation modeling support. I think that this functionality is very useful and I think that some of existing applications should implement it too (I wonder why it hasn't yet been implemented).

Bibliography

[1] AbbotWeb. Abbot web page, 2011.
`http://www.junit.org/` state from 2011-04-20.

[2] ArgoUMLWeb. ArgoUML web page, 2011.
`http://argouml.tigris.org/` state from 2011-04-16.

[3] FOWLER, M. *Destilované UML*. Prague : Grada Publishing, a.s., 2009. ISBN 978-80-247-2062-3.

[4] FOWLER, M. – NEUSTADT, I. *UML 2 and the Unified Process : Practical Object-Oriented Analysis and Design.* : Addison-Wesley Professional, 2005. ISBN 978-0321321275.

[5] GitWeb. GIT web page, 2011.
`http://git-scm.com/` state from 2011-05-11.

[6] JGraphWeb. JGraph web page, 2011.
`http://www.jgraph.com/` state from 2011-05-11.

[7] JUnitWeb. JUnit web page, 2011.
`http://www.junit.org/` state from 2011-04-16.

[8] MagicDrawWeb. MagicDraw web page, 2011.
`http://www.magicdraw.com/` state from 2011-05-11.

[9] MILNE, P. *Using XMLEncoder* [online]. 2010. [cit. 2011-01-29]. Available from: <`http://java.sun.com/products/jfc/tsc/articles/persistence4/`>.

[10] PECINOVSKÝ, R. *Návrhové Vzory*. Brno : Computer Press, a.s., 2007. ISBN 978-80-251-1582-4.

[11] PETRI, J. *NetBeans Platform 6.9 Developer's Guide*. Birmingham : Packt Publishing Ltd, 2010. ISBN 978-1-849511-76-6.

[12] SeleniumWeb. Selenium web page, 2011.
`http://seleniumhq.org/` state from 2011-04-20.

[13] sparxsystemsweb. Sparx Systems web page, 2011.
`http://www.sparxsystems.com/` state from 2011-03-29.

[14] StarUMLWeb. StarUML web page, 2011.
`http://staruml.sourceforge.net/` state from 2011-05-11.

[15] UMLWeb. UML web page, 2011.
`http://www.uml.org/` state from 2011-04-27.

[16] *JGraph 5 manual* [online]. 09 2009. [cit. 2011-03-03]. Available from: <`http://www.jgraph.com/downloads/jgraph/jgraphmanual.pdf`>.

Appendix A

Package Structure of Editor Module

You can see the package structure in this directory tree:

- cz.cvut.indepmod.classmodel
  - actions ..... Actions for buttons etc.
    - nbfolders ..... Actions which are registered in layer.xml
  - diagramdata ..... Classes which stores the information about the diagram
    - langs ..... Definition of language specific features (e.g. the data type names)
  - file ..... SaveCookie and file type association support
    - wizard ..... Wizard for creation of new class model file
  - frames
    - dialogs ..... GUI dialogs
      - factory ..... Factories for some dialogs
      - validation ..... Validation of values from dialogs
  - persistence ..... Layer for data saving
    - xml ..... implementation of persistence for saving into a XML file
      - delegate ..... XML Delegates for mapping into xml
  - resources ..... Bundle util
  - util ..... Utility classes
  - workspace ..... Main package containing the TopComponent with JGraph
    - cell ..... Cells, renderers, VertexViews, etc. for JGraph
      - components ..... Component for JGraph
      - model
        - classmodel .. Implementation of ClassModel API from API Module

Appendix B

Abstract Factory Code Example

```
1  public abstract class AbstractGUIFactory {
2
3    private static AbstactGUIFactory linuxFactory = null;
4    private static AbstractGUIFactory windowsFactory = null;
5
6    public static AbstractGUIFactory getFactory() {
7      String osName = System.getProperty("os.name");
8      if (osName.startsWith("Windows")) {
9        if (windowsFactory == null) {
10         windowsFactory = new WindowsGUIFactory();
11       }
12       return windowsFactory;
13     } else if (osName.equals("Linux")) {
14       if (linuxFactory == null) {
15         linuxFactory = new LinuxGUIFactory();
16       }
17       return linuxFactory;
18     } else {
19       //For example throw an exception
20     }
21   }
22
23   public abstract JButton createButton();
24   public abstract JLabel createLabel();
25 }
26
27 public class WindowsGUIFactory extends AbstractGUIFactory {
28
29   public JButton createButton() {
30     return new WindowsButton();
31   }
32
```

```
33    public JLabel createLabel() {
34      return new WindowsLabel();
35    }
36  }
37
38  public class LinuxGUIGactory extends AbstractGUIFactory {
39
40    public JButton createButton() {
41      return new LinuxButton();
42    }
43
44    public JLabel createLabel() {
45      return new LinuxLabel();
46    }
47  }
```

Appendix C

Graphic User Interface Test Cases

C.1 TC0 - Create new class diagram

Prerequisite: Open a project with a package.

- Right click on a package → click on 'new' → click on 'Other' → select 'IndepMod' in a list on the left → select 'New Class Diagram' in the list on the right → click 'Next' button
- *System will open a panel with field for the name of the diagram, selection of the diagram type and field with the desired language*
- Fill the form as follows:
  - Name: ClassModel
  - Select class model in the model type
  - Select Java in Language selection
  - Click on Finish
- *System will show an empty class diagram*

C.2 TC1 - Create business diagram

Prerequisite: Opened project with a package.

- Right click on a package → click on 'new' → click on 'Other' → select 'IndepMod' in a list on the left → select 'New Class Diagram' in the list on the right → click 'Next' button
- *System will open a panel with field for the name of the diagram, selection of the diagram type and field with the desired language*
- Fill the form as follows:
  - Name: BusinessModel
  - Select business model in the model type
  - Select Java in Language selection
  - Click on Finish
- *System will show an empty business diagram*

C.3 TC2 - Open the Tool Chooser

- Click on 'Window' in the maim menu and select 'Tool Chooser'
- *System will open the Tool Chooser panel with buttons for tool selection:*
  - *controll*
  - *class*
  - *interface*
  - *enumeration*
  - *relation*
  - *aggregation*
  - *composition*
  - *generalization*
  - *realization*

C.4 TC3 - Create new class

Prerequisite: Opened an empty class diagram

- Open the Tool Chooser [Extension Point: TC2]
- Activate the diagram workspace to register it in the Tool Chooser
- Select 'class' in the Tool Chooser
- *ToolChooser will show the selection*
- Click somewhere in the diagram workspace to add new class
- *System will add new class into the workspace with name 'Class1'*

C.5 TC4 - Create new interface

Prerequisite: Opened an empty class diagram

- Open the Tool Chooser [Extension Point: TC2]
- Activate the diagram workspace to register it in the Tool Chooser
- Select 'interface' in the Tool Chooser
- *ToolChooser will show the selection*
- Click somewhere in the diagram workspace to add new interface
- *System will add new interface into the workspace with name 'Interface1' and stereotype 'interface'*

C.6 TC5 - Create new enumeration

Prerequisite: Opened an empty class diagram

- Open the Tool Chooser [Extension Point: TC2]
- Activate the diagram workspace to register it in the Tool Chooser
- Select 'enumeration' in the Tool Chooser

- *ToolChooser will show the selection*
- Click somewhere in the diagram workspace to add new enumeration
- *System will add new enumeration into the workspace with name 'Enum1' and stereotype 'enumeration'*

C.7 TC6 - Open a class edit dialog

Prerequisite: Opened a class diagram with a class

- Right click on the class and select 'Edit'
- *System will open an edit dialog with fields:*
- *Name*
- *Data type*
- *Abstract checkbox*
- *Annotation list with buttons for creation, editing and deletion*
- *Attribute list with buttons for creation, editing and deletion*
- *Method list with buttons for creation, editing and deletion*
- *Buttons save/cancel*

C.8 TC7 - Add a class attribute

Prerequisite: Opened a class diagram with a class

- Open the class edit dialog [include TC6]
- Click on 'Add Attribute' button
- *System will open a dialog for attribute creation with fields:*
- *Name*
- *Data Type*
- *Visibility*
- *Annotation list with buttons for creation, editing and deletion*
- *Create button*
- Fill the fields in the dialog and click on 'Create'
- *System will add the attribute into the Attribute list in the class edit dialog and also into the class in the diagram*
- Click on 'Save' button and verify that the attribute was added properly

C.9 TC8 - Edit a class attribute

Prerequisite: Opened a class diagram with a class which has an attribute

- Open the class edit dialog [include TC6]
- Select an attribute in the attribute list and click on 'Edit Attribute' button
- *System will open a dialog for attribute creation with filled fields according to the selected attribute*

- Change some attribute fields and click on 'Create' button
- *System will change the attribute in the Attribute list in the class edit dialog and also in the class in the diagram*
- Click on 'Save' button and verify that the attribute was changed properly

C.10 TC9 - Remove a class attribute

Prerequisite: Opened a class diagram with a class which has an attribute

- Open the class edit dialog [include TC6]
- Select an attribute in the attribute list and click on 'Remove Attribute' button
- *System will remove the attribute from the Attribute list in the class edit dialog and also from the class in the diagram*
- Click on 'Save' button and verify that the attribute was removed properly

C.11 TC10 - Add a class method

Prerequisite: Opened a class diagram with a class

- Open the class edit dialog [include TC6]
- Click on 'Add Method' button
- *System will open a dialog for method creation with fields:*
- *Name*
- *Data Type*
- *Visibility*
- *Static checkbox*
- *Abstract checkbox*
- *Attribute list with buttons for creation, editing and deletion*
- *Save/Cancel button*
- Fill the fields in the dialog and click on 'Save'
- *System will add the method into the method list in the class edit dialog and also into the class in the diagram*
- Click on 'Save' button and verify that the method was added properly

C.12 TC11 - Edit a class method

Prerequisite: Opened a class diagram with a class which has a method

- Open the class edit dialog [include TC6]
- Select a method in the method list
- Click on 'Edit Method' button
- *System will open a dialog for method creation with filled fields according to the method*
- Change the fields in the dialog and click on 'Save'

- *System will change the method in the method list in the class edit dialog and also in the class in the diagram*
- Click on 'Save' button and verify that the method was changed properly

C.13 TC12 - Remove a class method

Prerequisite: Opened a class diagram with a class which has a method

- Open the class edit dialog [include TC6]
- Select a method in the method list
- Click on 'Delete Method' button
- *System will remove the method from the method list in the class edit dialog and also from the class in the diagram*
- Click on 'Save' button and verify that the method was removed properly

C.14 TC13 - Add a class annotation

Prerequisite: Opened a class diagram with a class

- Open the class edit dialog [include TC6]
- Click on 'Add Annotation' button
- *System will open a dialog for annotation creation with fields:*
- *Name*
- *Value list with buttons for creation and deletion*
- *Create button*
- Add an annotation value [Extension point: TC16]
- Remove an annotation value [Extension point: TC17]
- Fill the fields in the dialog and click on 'Save'
- *System will add the annotation into the annotation list in the class edit dialog and also into the class in the diagram*
- Click on 'Save' button and verify that the annotation was added properly

C.15 TC14 - Edit a class annotation

Prerequisite: Opened a class diagram with a class which has an annotation

- Open the class edit dialog [include TC6]
- Select an annotation in the annotation list
- Click on 'Edit Annotation' button
- *System will open a dialog for annotation creation with filled fields according to the selected annotation*
- Add an annotation value [Extension point: TC16]
- Remove an annotation value [Extension point: TC17]
- Change the fields in the dialog and click on 'Save'

- *System will change the annotation in the annotation list in the class edit dialog and also in the class in the diagram*
- Click on 'Save' button and verify that the annotation was changed properly

C.16 TC15 - Remove a class annotation

Prerequisite: Opened a class diagram with a class which has an annotation

- Open the class edit dialog [include TC6]
- Select an annotation in the annotation list
- Click on 'Delete Annotation' button
- *System will remove the annotation from the annotation list in the class edit dialog and also from the class in the diagram*
- Click on 'Save' button and verify that the annotation was removed properly

C.17 TC16 - Add an annotation value

Prerequisite: Opened a dialog for annotation creation/editing.

- Click on 'Add Value' button
- *System will open a dialog for attribute value creation with fields:*
  - *Name*
  - *Value list with delete button*
  - *Value text field with add button*
  - *Create button*
- Fill these fields and click the Create button
- *System will add an annotation value into the annotation value list*

C.18 TC17 - Remove an annotation value

Prerequisite: Opened a dialog for annotation creation/editing.

- Select an annotation value in the annotation value list and click on Delete value button
- *System will remove the annotation value from the annotation value list*

C.19 TC18 - Add a relation

Prerequisite: Opened a dialog with at least two classes.

- Open the Tool Chooser [Extension Point: TC2]
- Select the relation in the Tool Chooser
- *Tool Chooser will show the selection and classes in the diagram will have blue rectangles indicating the ability to join the relation*

- Add the relation between two classes by clicking and holding the left mouse button on a class, moving into another class and releasing the left mouse button
- *System will add the relation between desired classes*

C.20 TC19 - Add an aggregation

Prerequisite: Opened a dialog with at least two classes.

- Open the Tool Chooser [Extension Point: TC2]
- Select the aggregation in the Tool Chooser
- *Tool Chooser will show the selection and classes in the diagram will have blue rectangles indicating the ability to join the aggregation*
- Add the aggregation between two classes by clicking and holding the left mouse button on a class, moving into another class and releasing the left mouse button
- *System will add the aggregation between desired classes*

C.21 TC20 - Add a composition

Prerequisite: Opened a dialog with at least two classes.

- Open the Tool Chooser [Extension Point: TC2]
- Select the composition in the Tool Chooser
- *Tool Chooser will show the selection and classes in the diagram will have blue rectangles indicating the ability to join the composition*
- Add the composition between two classes by clicking and holding the left mouse button on a class, moving into another class and releasing the left mouse button
- *System will add the composition between desired classes*

C.22 TC21 - Add a generalization

Prerequisite: Opened a dialog with at least two classes.

- Open the Tool Chooser [Extension Point: TC2]
- Select the generalization in the Tool Chooser
- *Tool Chooser will show the selection and classes in the diagram will have blue rectangles indicating the ability to join the generalization*
- Add the generalization between two classes by clicking and holding the left mouse button on a class, moving into another class and releasing the left mouse button
- *System will add the generalization between desired classes*

C.23 TC22 - Add a realisation

Prerequisite: Opened a dialog with at least one class and one interface.

- Open the Tool Chooser [Extension Point: TC2]

- Select the realisation in the Tool Chooser
- *Tool Chooser will show the selection and classes in the diagram will have blue rectangles indicating the ability to join the realisation*
- Add the realisation between the class and interface by clicking and holding the left mouse button on the class, moving into the interface and releasing the left mouse button
- *System will add the realisation between the class and the interface*

C.24 TC23 - remove a relation

Prerequisite: Opened a dialog with a realation

- Right click on the relation and select delete
- *System will remove the relation*

C.25 TC24 - remove an aggregation

Prerequisite: Opened a dialog with an aggregation

- Right click on the aggregation and select delete
- *System will remove the aggregation*

C.26 TC25 - remove a composition

Prerequisite: Opened a dialog with a composition

- Right click on the composition and select delete
- *System will remove the composition*

C.27 TC26 - remove a generalization

Prerequisite: Opened a dialog with a generalization

- Right click on the generalization and select delete
- *System will remove the generalization*

C.28 TC27 - remove a realisation

Prerequisite: Opened a dialog with a realisation

- Right click on the realisation and select delete
- *System will remove the realisation*

C.29 TC28 - Edit a relation

Prerequisite: Opened a dialog with a relation

- Right click on the relation and select edit
- *System will open a relation edit dialog with fields:*
  - *Name*
  - *Source Cardinality*
  - *Target Cardinality*
  - *Relation type (Unidirectional, Bidirectional)*
  - *Show arrows checkbox*
  - *Name along the edge checkbox*
  - *Save and Cancel buttons*
- Fill the fields and click Save button
- *System will update the relation*

C.30 TC29 - Edit an aggregation

Prerequisite: Opened a dialog with an aggregation

- Right click on the aggregation and select edit
- *System will open an aggregation edit dialog with fields:*
  - *Name*
  - *Source Cardinality*
  - *Target Cardinality*
  - *Relation type (Unidirectional, Bidirectional)*
  - *Show arrows checkbox*
  - *Name along the edge checkbox*
  - *Save and Cancel buttons*
- Fill the fields and click Save button
- *System will update the aggregation*

C.31 TC30 - Edit a composition

Prerequisite: Opened a dialog with a composition

- Right click on the composition and select edit
- *System will open a composition edit dialog with fields:*
  - *Name*
  - *Source Cardinality*
  - *Target Cardinality*
  - *Relation type (Unidirectional, Bidirectional)*
  - *Show arrows checkbox*

  - *Name along the edge checkbox*
  - *Save and Cancel buttons*
- Fill the fields and click Save button
- *System will update the composition*

C.32 TC31 - Save and Load a diagram

Prerequisite: Open an empty class diagram

- Create a class [INCLUDE TC3]
- Create another three classes [INCLUDE TC3]
- Add an interface [INCLUDE TC4]
- Add an enumeration [INCLUDE TC5]
- Add a class attribute [INCLUDE TC7]
- Add a class method [INCLUDE TC10]
- Add a class annotation [INCLUDE TC13]
- Add a relation [INCLUDE TC18]
- Add an aggregation [INCLUDE TC19]
- Add a composition [INCLUDE TC20]
- Add a generalization [INCLUDE TC21]
- Add a realisation [INCLUDE TC22]
- Click on the File in the main menu and select Save
- *System will save the diagram (The bold font of the diagram's name will be changed to the normal font)*
- Close the diagram
- *System will close the diagram*
- Open the diagram
- *System will open the diagram*
- Verify that all elements were loaded properly