ENGLISH VERSION

User Guide

PyUML for Eclipse

A UML Roundtrip Tool for Python

Version 1.2

Authors: Martin Dittmar

Translators: Martin Dittmar

Translation Status: incomplete

Inhaltsverzeichnis

This User's Guide leads through the installation und usage of the PyUML Roundtrip Tool.

1 System requirements

PyUML is a Eclipse Plugins and this runnable on all Platforms supported by Eclipse. A list of all platforms is in the download area of Eclipse[1].

Platform-independent requirements are:

Hardware

- current CPU, e.g. 2-GHz
- 512 MB RAM (recommended 1 GB or more)
- ca. 200 MB free space on hard disk

Software

- JAVA Runtime Environment (JRE) with version 1.5 or higher [2]
- Python Runtime Environment with version 2.4 or higher[3]
- graphical interface
- any Eclipse, version 3.3 oder higher
- PyDev, version 1.4.2 or higher (can be installed automatically as dependency in installation process)

2 Installation

Although there are several possibilities to install PyUML, it is recommended to use the *Eclipse Update Manager*.

2.1 Use of the Eclipse Update Manager

Use the PyUML Update Site in the Update Manager of Eclipse:

Go through the menu:

Help->Software Update->Available Software->Add Site

Add the following Addresses:

http://eclipse-pyuml.sourceforge.net/update_site

and (if pyDev is not already present):

http://pydev.sourceforge.net/updates/

Now you can choose *eclipse-pyUml* and click on *Install*. All dependencies, incl. *PyDev*, should now be resolved.

2.2 Installation as complete package

Hint: A complete package is only available for PyUML 1.0 with Eclipse 3.2. Please use the Update Site instead!

TRANSLATION MISSING!

1 http://www.eclipse.org/downloads/
2 Eine aktuelle JAVA-Laufzeitumgebung kann unter http://java.sun.com heruntergeladen werden.
3 Eine aktuelle Python-Laufzeitumgebung kann unter http://www.python.org/ heruntergeladen werden.

Die wahrscheinlich einfachste, aber auch unflexibelste Variante der Installation von PyUML ist das Herunterladen und Entpacken einer kompletten Eclipse-Installation, die PyUML und alle benötigten Plugins enthält. Diese Variante wird für Windows und Linux (jeweils Versionen für die 32-Bit-x86-Architektur) angeboten.

Das PyUML-Komplettpaket kann als ZIP-gepackte Datei auf der folgenden Seite heruntergeladen werden:

- http://sourceforge.net/projects/eclipse-pyuml/

Nach dem Herunterladen kann die Zip-Datei entpackt werden, wodurch ein *eclipse*-Verzeichnis erzeugt wird, das alle benötigten Plugins enthält. Eclipse lässt sich nun durch Doppelklick auf die Datei *eclipse* im Eclipse-Ordner starten.

2.3 Installation as Plugin Package

Hint: A plugin package is only available for PyUML 1.0 with Eclipse 3.2. Please use the Update Site instead!

TRANSLATION MISSING!

Falls es schon eine Eclipse-Installation gibt, die durch PyUML erweitert werden soll, oder auf Plattformen, für die kein Komplettpaket angeboten wird, empfiehlt sich die Installation von PyUML als Plugin-Paket in eine existierende Eclipse-Installation hinein.

Falls Eclipse noch nicht installiert ist, kann es von http://www.eclipse.org/downloads/ heruntergeladen werden. Ausreichend ist der relativ kleine Download *Eclipse IDE for Java Developers*. Nach erfolgter Installation können nun alle für PyUML nötigen Features nachinstalliert werden.

Dazu lädt man das Paket auf der Seite

- http://sourceforge.net/projects/eclipse-pyuml/

herunter. Nach dem Entpacken kopiert man die Inhalte aus dem *plugin*- und aus dem *features*-Ordner in die entsprechenden Ordner der Eclipse-Installation. Beim nächsten Start von Eclipse wird das PyDev-Plugin automatisch gestartet.

3 Configuring PyUML

Although PyUML has no own configuration page in the Eclipse Preferences, it needs some change on the Eclipse configuration to work smoothly.

Changing the heap size in eclipse.ini

PyUML is because of the usage of the Eclipse Uml2Tools very greedy for RAM memory. If Eclipse cannot allocate enough memory, it just shuts down. Because of that the available memory for Eclipse has to be increased. For that, the file *eclipse.ini* has to be edited.

The file eclipse.ini is located in the Eclipse Programs folder.

Please add the folowing lines to *eclipse.ini*:

```
-Xmx800M
-XX:MaxPermSize=500M
```

Configuring PyDev

Before Eclipse can be used for Python development the *PyDev*-Plugin has to be configured. PyDev has to know where to find it's runtime environment.

To configure PyDev , choose in Eclipse *Window→Preferences→PyDev* and then *Interpreter-Python*. Now in the window *Python Interpreters* choose *new* and type/choose the path of your python interpreter. Under Linux this is usually */usr/bin/python*[4], under Windows this is the Path of python.exe, usually under *C:\Program Files*. By typing OK, all needed libraries are found and PyDev is configured. Now you can create PyDev projects and execute Python code.

4 Creation of a PyDev Project

Because PyUML is based on PyDev, there must exist a PyDev (Python) project in order to work with UML diagrams. If there is no PyDev project, create a new one under *File→New→Project->PyDev-Project*. In this process you have to define a sources folder which is also used for roundtrip synchronization.

The files created by PyUML namely the UML model, a model backup, the Diagramm and the view files and the PyUML profile, are saved in the folder *PyUML* directly below the project folder. You can normally ignore this directory, as all functionality UML works automatically without choosing a file in this directory. But this directory should be added to version control (if needed), if it should be available to all developers.

5 Basic functions of PyUML

After Installation of PyUML there are 4 new symbols in the toolbars:

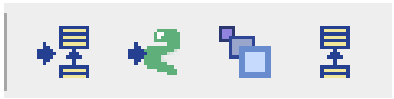

Image 1: Plugin icons of PyUML

These Icons are for steering of PyUML. The description of the symbols from left to right is as follows:

1. **Model Synchronization**

PyUML is looking in the PyDev source folder for Python source code and is also looking for a previously created UML model. If there is no model, a new model will be created according to the Python code. If you want to start from scratch with the model, use this button to create an empty model that can be used for Code generation.

After every code change and before changing the model a model synchronization has to be executed to avoid inconsistencies.

The created/synchronized model will be opened automatically.

2. **Code Synchronization**

Just like model synchronization, this is to change or create Python code according to the model. After changing the model this must be run to avoid inconsistencies.

3. **Views Management**

With this button, views can be created, opened, edited and deleted. Views are user-defined sights of the UML model that allow displaying classes of different packages and relations between them, which is not possible in the normal model views of the Eclipse Uml2Tools Editor.

Views are kept synchronous with the “main” model, so that editing classes can take place in the main model or in a view. A detailed description can be found below in the User Guide.

4. **Opening an already present UML model**

4 You can find out which interpreter is relly used by typing „*which python*“ on the command line.

If the UML model is already synchronous with the Python code, it can be opened without synchronization here. This is useful with very big projects, where a synchronization run can take some time.

Note: alle described function look for PyDev Projects in the Eclipse Workspace. If there is more than one PyDev project, the user is asked which to use, with the last choice pre-selected.

All these functions are also present in the context menu of the project by right-clicking the project in the package explorer, which may be faster than using the buttons.

6 Usage of the UML Editor

TRANSLATION MISSING! Please use the Eclipse Uml2Tools documentation!

Der PyUML UML-Editor basiert auf dem Eclipse-Uml2Tools-Editor. Nach einer Modellsynchronisation wird automatisch der Editor geöffnet. Im Folgenden wird beispielhaft mit einem leeren Projekt begonnen, für das ein neues Modell erstellt wird. Dazu wird ein neues PyDev-Projekt namens *PyUmlTest* erstellt und die Modellsynchronisation angestoßen, wonach der UML-Editor mit einem leeren Diagramm geöffnet wird:

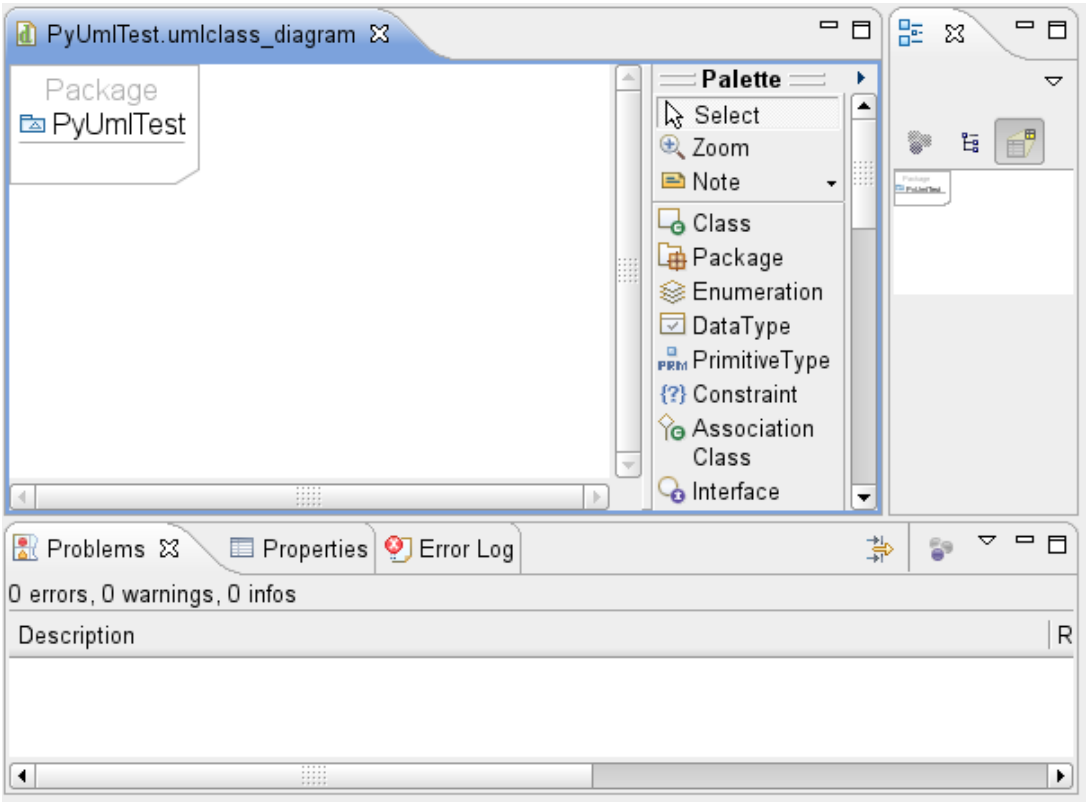

Abbildung 2: Leeres UML-Klassendiagramm

Auf der linken Seite befindet sich die UML-Diagrammansicht, die bis auf den Modellnamen in der linken oberen Ecke leer ist. Der Name des Modells entspricht dabei immer dem Namen des Projekts. Teil dieser Ansicht ist die Palette, über die sich die Modellelemente auswählen lassen. Die Palette kann über den nach rechts zeigenden Pfeil am oberen Ende ausgeblendet werden.

Rechts daneben befindet sich der *Outline*-View, der eine Übersicht über das gesamte Diagramm und den aktuellen Ausschnitt zeigt, wenn das Diagramm nicht auf den Bildschirm passt. Über den *Outline*-View lässt sich die aktuelle Ansicht außerdem bequem verschieben.

Unten ist der *Problems View* geöffnet, der die Ergebnisse der Modellvalidierung anzeigt, im Augenblick aber mangels Modellelementen, die validiert werden könnten, leer ist.

In der der unteren Ansicht befindet sich außerdem ein Reiter *Properties*, in dem sich Eigenschaften von Modellelementen, die nicht in der GUI nicht angezeigt werden, bearbeiten lassen. Der *Properties View* lässt sich auch über das Kontextmenü eines Modellelements erreichen.

Die weitere Bedienung des Editors wird beispielhaft durch Erstellen eines Diagramms erklärt. Als erstes werden nun dem Diagramm zwei Pakete hinzugefügt, in denen einige Klassen erstellt werden sollen. Das Hinzufügen von Paketen kann, wie bei grundsätzlich allen Modellelementen, über die Auswahl in der Palette und Platzierung des Elements erfolgen. Eine andere Möglichkeit ist, den Mauszeiger an die Stelle zu positionieren, wo das Element eingefügt werden soll. Nach ungefähr einer Sekunde wird eine grafische Liste mit allen an dieser Position verfügbaren Elementen angezeigt. Fährt man mit der Maus über ein Symbol in der Liste, wird eine Kontexthilfe angezeigt:

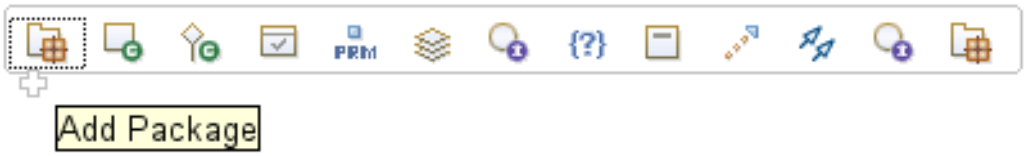

Abbildung 3: Kontextabhängige Palette im UML-Diagramm

Hier kann das Paketsymbol ausgewählt werden, wodurch ein neues Paket eingefügt wird. Der Fokus befindet sich nun in der Eingabe des Paketnamens, das jetzt manuell von *package* auf *backend* geändert. Das ganze wird für ein zweites Paket namens *gui* wiederholt. Die beiden Pakete werden wie folgt dargestellt:

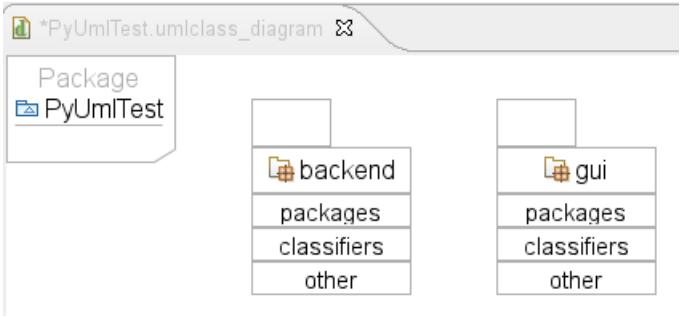

Abbildung 4: UML-Editor: Erstellung von Paketen

Die Pakete lassen sich durch Ziehen mit der Maus beliebig verschieben. Will man nun den Inhalt des Pakets *backend* bearbeiten, klickt man mit der Maus doppelt auf den oberen Teil des Pakets, wonach sich eine neue Ansicht mit dem entsprechenden Paket als Basis öffnet.

Darin können nun als Beispiel die Klassen *FTPConnection*, *ConnectionCollector* und *ConnectionHelper* sowie das Interface *Connection* eingefügt werden. Im Diagramm wird das Interface *Connection* als Kreis dargestellt. Da dem Interface Methoden hinzufügt werden sollen, muss das Interface als Klasse dargestellt werden. Dazu wählt man im Kontextmenü des Interfaces den Punkt *Show As Class*. Im Beispiel soll *FTPConnection* eine Ausprägung von *Connection* sein, weshalb eine Vererbung zwischen den Klassen eingefügt wird.

Nun wird eine Assoziation von *FTPConnection* zu *ConnectionHelper* modelliert. Außerdem soll es eine navigierbare 1-zu-n-Assoziation von *ConnectionCollector* zu *Connection* geben, womit ausgedrückt wird, dass in *ConnectionCollector* vorhandene Verbindungen gespeichert und gefunden werden können. Die 1-zu-n-Assoziation wird dadurch realisiert, dass in dem Textfeld des Assoziationsziels (standardmäßig *dst*) *[1..*]* hinzugefügt wird. Der UML-Editor verwertet dies automatisch als Quantität und fügt sie entsprechend im Diagramm ein.

Danach werden in den Klassen und dem Interface einige Attribute und Methoden modelliert. Die Klasse *ConnectionHelper* erhält dabei statische Attribute und Hilfsmethoden, die ohne Instantiierung von *ConnectionHelper* statisch aufrufbar sein sollen. Dazu wird das Attribut oder die Methode markiert und im *Properties View* der Wert *isStatic* auf *true* gesetzt. Bei den Methoden

muss das static-Attribut nicht manuell gesetzt werden. Stattdessen kann einfach der erste Parameter *self* weggelassen werden, der sonst bei allen nicht-statischen Methoden eingetragen werden muss. Bei der Codeerzeugung wird diese Methode dann automatisch als statisch interpretiert.

Das fertige Diagramm kann nun wie folgt aussehen:

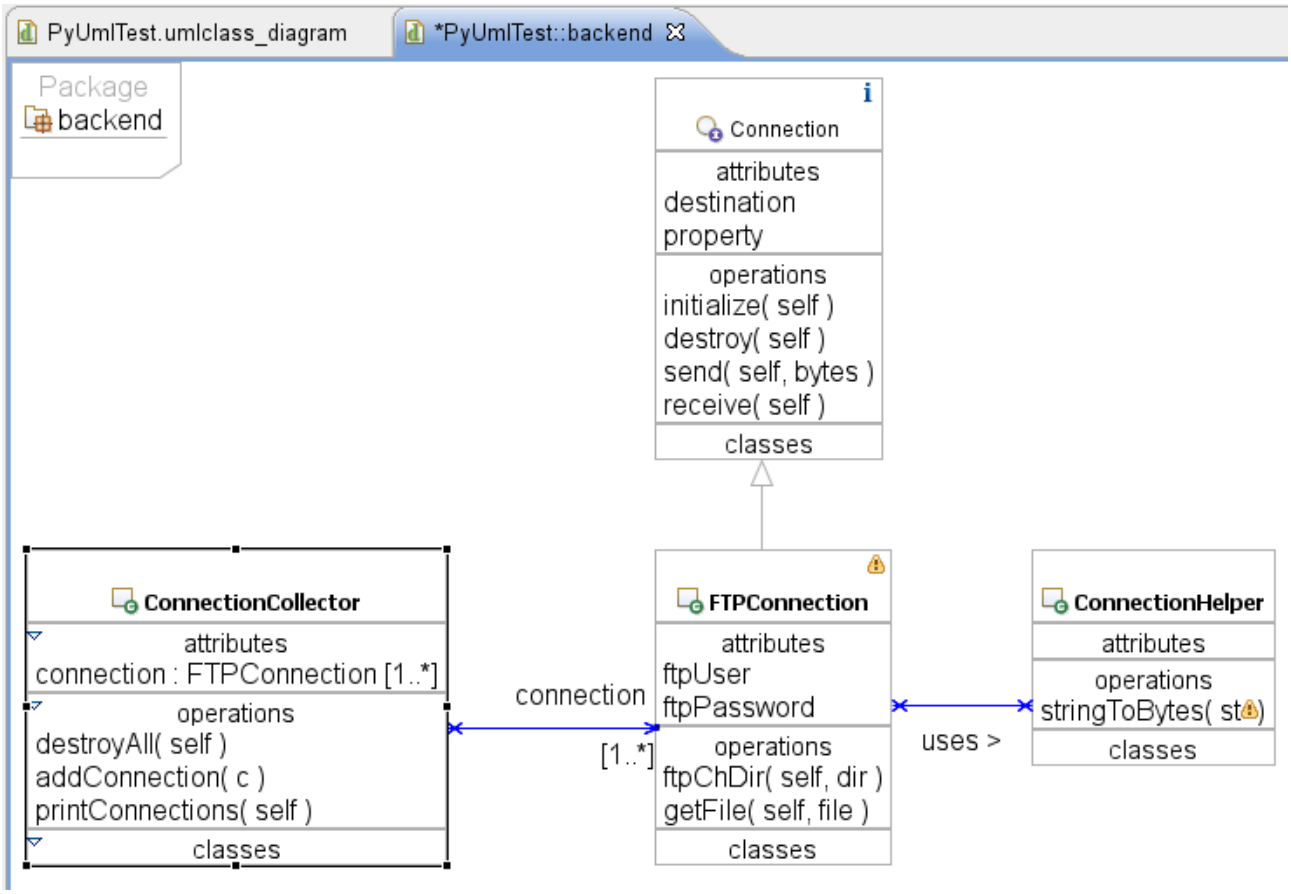

Abbildung 5: Beispielklassen im Paket backend

Auffällig ist, dass *ConnectionCollector* durch die navigierbare Assoziation automatisch ein Attribut namens *Connection* vom Typ *FTPConnection* erhält, in dem auch die Quantität des Attributs eingetragen ist. Dieses Attribut wird später auch in Python-Code umgesetzt, auch wenn der Typ und die Quantität dabei ignoriert werden.

Außerdem ist zu beobachten, dass bei *Connection* ein blaues *i* und bei *ConnectionHelper* ein gelbes Ausrufezeichen angezeigt wird. Das sind Informationen aus der Live-Validation, die später genauer betrachtet und im Augenblick ignoriert werden.

Weitere Bedienungshinweise

Um den Umfang des Uml2Tools-Editors voll auszunutzen, wird hier auf einige weitere Funktionen hingewiesen.

- Sichtbarkeitsmodifikatoren können durch ein Zeichen direkt vor dem Methodennamen angegeben werden. Die Zeichen entsprechen dem UML-Standard: „-“ für *private*, „#“ für *protected*, „~“ für *package*, „+“ oder kein Zeichen für *public*. Eine alternative Eingabe durch den *Properties View* ist auch möglich.
- Auch bei der Definition von Attributen, Methoden und Assoziationsenden können alle in der jeweiligen UML-Syntax festgelegten Eigenschaftswerte direkt im jeweiligen Namen eingetragen werden; sie werden entsprechend vom UML-Editor erkannt. Beispielweise können Quantitäten in eckigen Klammern hinter einem Namen eingetragen werden.

- Will man einzelne Elemente hervorheben, kann man über die Werkzeugleiste Formatierungen wie Schriftgröße, Hintergrundfarbe und Linienstile verändern.
- Die Klassendetails können über das Kontextmenü einer Klasse im Punkt *Filters→Show/Hide Compartments* ein- und ausgeblendet werden.
- Über den Punkt *Arrange All* in der Werkzeugleiste lassen sich alle Elemente des Diagramms hierarchisch geordnet neu ausrichten.
- Elemente können ohne Einrastpunkte verschoben werden, wenn nach Beginn des Verschiebevorgangs die *Alt*-Taste gedrückt gehalten wird.
- Bei Verbindungslinien zwischen Klassen wie z.B. Assoziationen kann die Position der Endpunkte geändert werden. Dazu sucht man mit der Maus den Endpunkt, an dem der Mauszeiger zu einem Kreuz wird und kann ihn nun verschieben. Verbindungslinien können beliebig oft ihre Richtung ändern; will man eine Linie um eine Ecke herumführen, kann man mit der Maus den neu zu erstellenden Eckpunkt auf der Linie anklicken und verschieben.

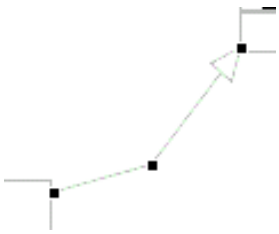

Abbildung 6: Einfügen eines Eckpunktes

7 Roundtrip Functionality

TRANSLATION MISSING!

Die Roundtrip-Fähigkeit beinhaltet die Anpassung und Erstellung von Code anhand des UML-Modells und umgekehrt. Dabei werden vorhandene Code- und Modellelemente nicht einfach ersetzt, sondern nur die entsprechenden Änderungen „eingepflegt“. Die Roundtrip-Funktionalität ist je Richtung einzeln aufrufbar, indem die beschriebenen Funktionen für die Modellsynchronisation bzw. die Codesynchronisation aufgerufen werden. Sie soll jetzt anhand des im vorigen Kapitels erstellten Modells demonstriert werden.

Da noch kein Python-Code vorhanden ist, wird die Codesynchronisation verwendet, um ein Codegerüst zu erstellen. Dazu kann der Button mit dem Schlangensymbol gedrückt werden.

Nach der Synchronisation befinden sich im *src*- Verzeichnis des Projekts die erstellten Pakete und Klassen, wie im Bild rechts dargestellt ist.

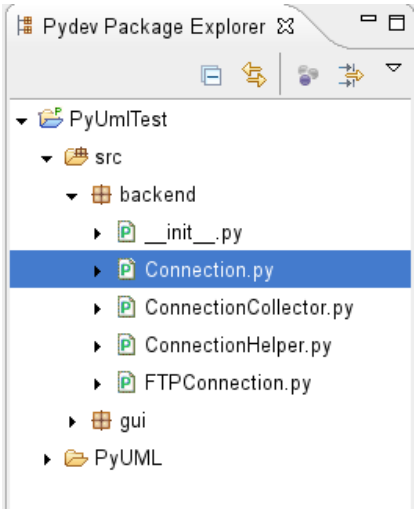

Abbildung 7: Automatisch erstellte Pakete und Klassen

Weil Python keine Interfaces kennt, wurde das Interface als Klasse realisiert, wobei im Code vermerkt wurde, dass es sich um ein Interface handelt. Auch die Vererbungen, Methoden und Attribute sowie die nötigen *import*-Statements wurden korrekt eingetragen.

Assoziationen werden im *DocString*-Kommentar einer Klasse zusammenfassend dargestellt. Statische Methoden werden mit einem *staticmethod*-Aufruf versehen, damit sie im Klassenkontext ausführbar sind:

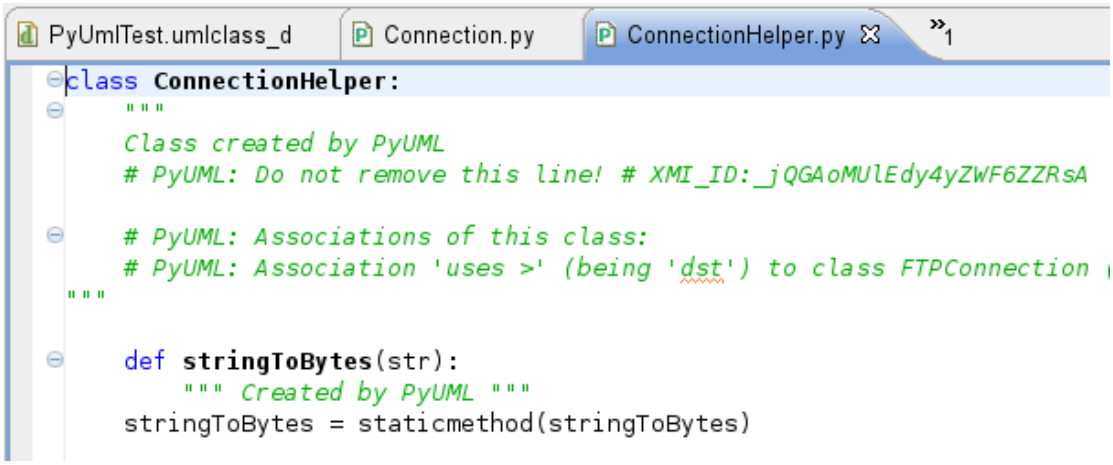

Abbildung 9: Automatisch erstellte Klasse ConnectionHelper mit Darstellung der Assoziationen und mit statischer Methode

Abbildung 8: Automatisch aus Interface erstellte Klasse Connection

Der erstellte Code kann nun im PyDev-Code-Editor bearbeitet werden. Dazu wird beispielhaft die Methode *ConnectionHelper* in ein neues Paket *helpers* verschoben. Außerdem werden einige Methoden und Attribute gelöscht und andere hinzugefügt. Einige Attribute bekommen zusätzlich einen Standardwert zugewiesen.

Nach einer Modellsynchronisation werden die vorgenommenen Änderungen entsprechend in das Modell übertragen:

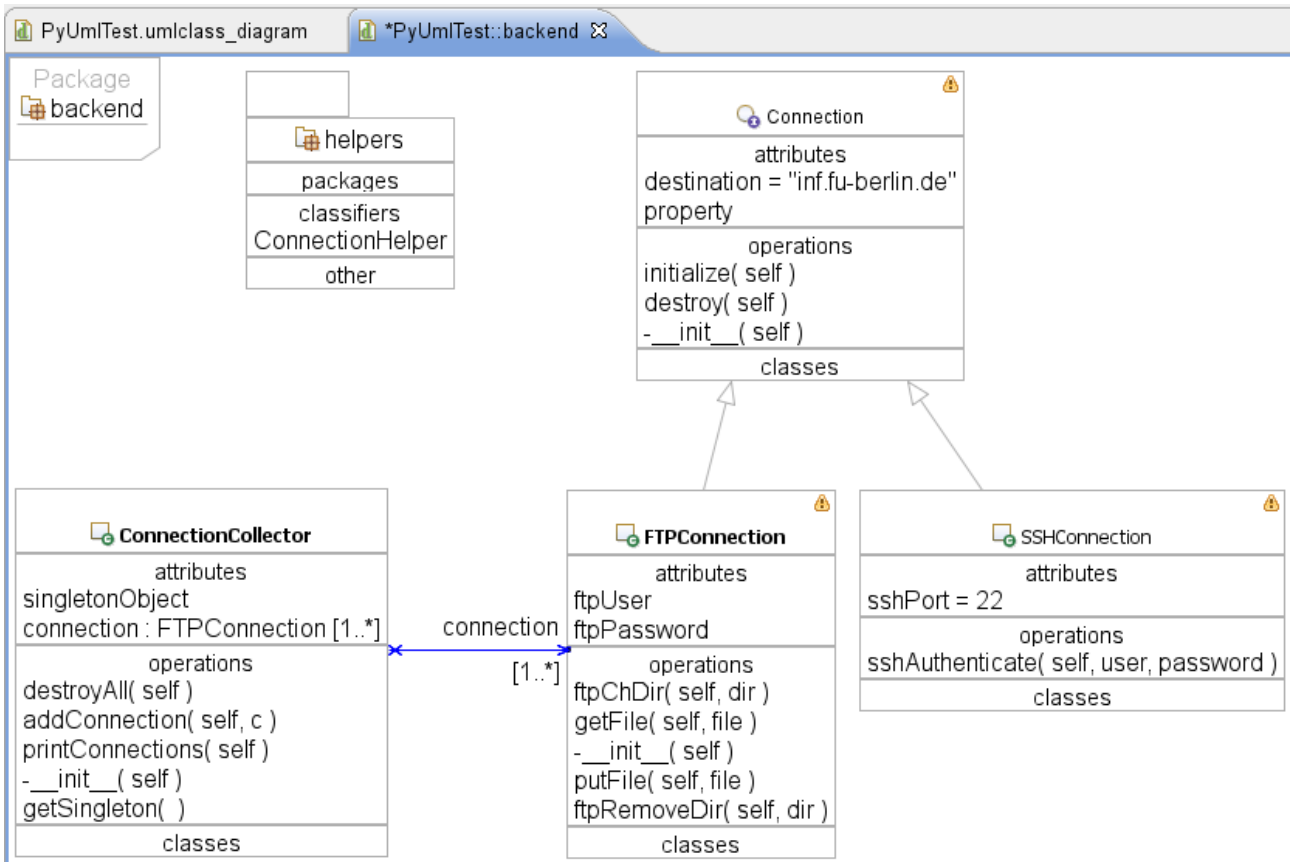

Abbildung 10: Modell nach Synchronisation

Die Klasse *ConnectionHelper* wurde in der Abbildung korrekt nach *helpers* verschoben, auch die

anderen Änderungen wurden erkannt und entsprechend eingefügt. Die Assoziation zwischen *FTPConnection* und *ConnectionHelper* wird nun aber nicht mehr angezeigt, weil sich *ConnectionHelper* in einem anderen Paket befindet. Um diese Assoziation sichtbar zu machen, könnten *Views* verwendet werden, auf die später eingegangen wird.

Auf die demonstrierte Weise kann eine Synchronisation zwischen Code und Modell immer gewährleistet werden.

8 Live Validation

Please note that Live Validation is only available in PyUML 1.0. In the current version Live Validation was disabled, because it would need patching the current Uml2Tools Editor!

TRANSLATION MISSING!

Live-Validation ist die während der Bearbeitung des Modells stattfindende automatische Überprüfung des Modells mit entsprechenden Meldungen für den Benutzer. Es gibt 3 Kategorien von Meldungen:

- *Error*: Das Modell enthält Fehler, die eine Codesynchronisation unmöglich machen. Das ist bei PyUML nur der Fall, wenn es mehrere Methoden mit gleichem Namen innerhalb einer Klasse gibt (Überladung), was in Python nicht sinnvoll realisiert werden kann.
- *Warning*: Der Benutzer soll auf bestimmte Inkonsistenzen hingewiesen werden, die eine Codeerstellung nicht verhindern, deren Umsetzung in Python aber nicht eindeutig ist. Das ist der Fall, wenn ein Sichtbarkeitsmodifikator in UML nicht aus dem Namen eines Elements hervorgeht (z.B. bei fehlenden Unterstrichen für ein privates Attribut), oder wenn die Methodenparameter nicht mit dem UML-*static*-Attribut konsistent sind. In diesen und anderen Fällen wird der Benutzer darauf hingewiesen, welcher der Werte für die Codeerzeugung verwendet wird, oder ob ein gewählter Wert überhaupt verwendet werden kann. Der Benutzer kann so entscheiden, ob er die Warnung ignoriert oder entsprechende Änderungen vornimmt.
- *Info*: Es wurde keine Inkonsistenz festgestellt, aber der Benutzer soll trotzdem über eine bestimmte Realisierung bei der Codeerstellung benachrichtigt werden, z.B. im Fall von Interfaces, die als Python-Klassen realisiert werden müssen.

Die Validierungsmeldungen werden sowohl im *Problems View* mit einem entsprechenden Text und dem verursachenden Element angezeigt, als auch im Diagramm selbst in Form von einem roten Kreuz für *Error* (), einem gelben Ausrufezeichen für *Warning (*) und einem blauen „i“ für *Info(*).

Wird beispielsweise eine neue Methode in eine Klasse eingefügt, so erzeugt sie sofort eine Warnung, weil sie noch keine definierten Parameter definiert hat, also auch noch keinen *self*-Parameter an erster Stelle. Methoden ohne self werden in Python als statische Methoden realisiert. Da der UML-seitige Wert *isStatic* aber noch auf *false* steht, wird eine Inkonsistenz erkannt und angezeigt, dass bei der Codeerstellung die Parameterdefinition Vorrang hat vor dem UML-seitigen Wert. Bei der Codeerzeugung würde also ein entsprechender *staticmethod*-Aufruf erzeugt.

Der Benutzer kann nun als ersten Parameter *self* eingeben oder den *isStatic*-Wert im *Properties View* ändern. In beiden Fällen verschwindet die Warnung. Er kann die Warnung aber auch ignorieren. Nach einem kompletten Roundtrip-Zyklus würde die Warnung ohnehin verschwinden, weil die in Python als statisch realisierte Methode bei einer Modellsynchronisation als statisch erkannt und das Modell angepasst würde, wodurch das Diagramm wieder konsistent wäre.

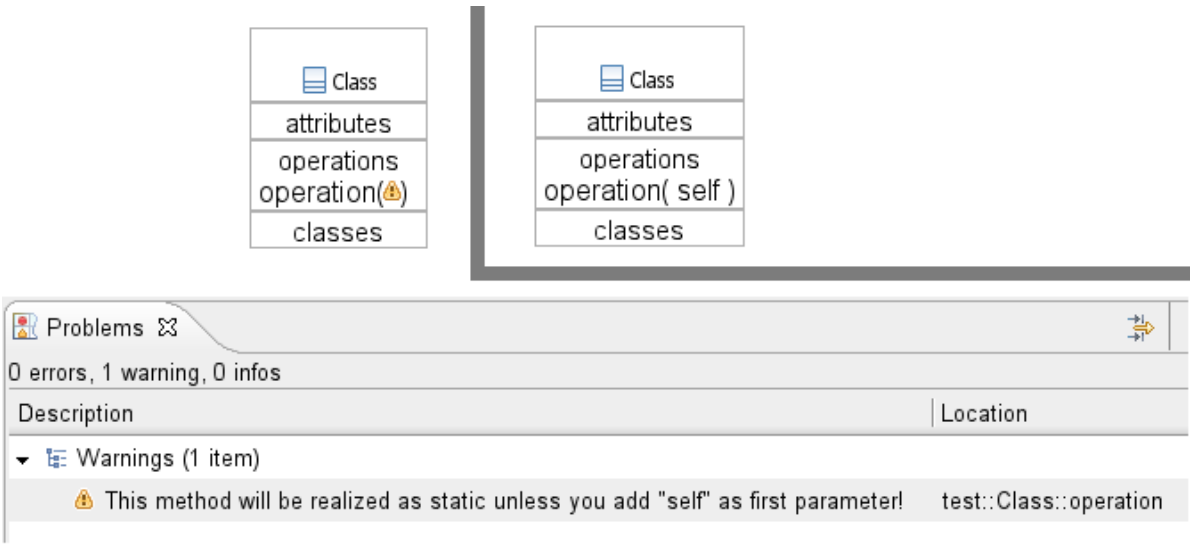

Abbildung 11: Live-Validation: Klasse mit angezeigter Warnung und zugehörigem Problems View und Klasse mit verschwundener Warnung

9 Views

TRANSLATION MISSING!

Ein View ist eine Teilansicht eines UML-Modells, die eine benutzerdefinierte Auswahl von Klassen unabhängig von Paketebenen enthält. Nur mit Views können Beziehung zwischen Klassen aus verschiedenen Paketen modelliert werden. Außerdem können Views verwendet werden, um bestimmte Zusammenhänge zu veranschaulichen, beispielsweise die Darstellung aller aktuellen Kindklassen einer Klasse.

Es können beliebig viele Views von einem Modell angelegt werden. Der View und das Modell werden dabei automatisch synchron gehalten, so dass Modelländerungen wahlweise am Modell selbst oder in einem View stattfinden können.

Um Views zu verwalten, ruft man die *Manage Views*-Funktionalität auf, wahlweise durch Klick auf das Icon in der Werkzeugleiste oder über das Kontextmenü des entsprechenden Projekts.

Nun wird der Dialog zur Verwaltung von Views angezeigt. Oben sind alle schon vorhandenen Views dargestellt, darunter gibt es entsprechende Buttons zum Öffnen, zur Erstellung, zur Bearbeitung und zum Löschen eines Views. Da noch keine Views erstellt wurden, ist nur der *Create New View*-Button auswählbar:

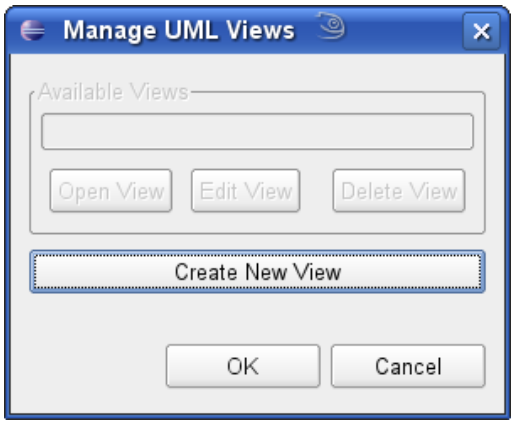

Abbildung 12: Initiale Views-Verwaltung

Wird dieser Button aktiviert, wird ein neuer Dialog geöffnet, der den Benutzer auffordert, den View zu benennen und die einzubeziehenden Klassen auszuwählen. Dabei kann man auch ganze Pakete

an- und abwählen, wodurch rekursiv alle enthaltenen Klassen und Pakete automatisch an- und abgewählt werden. Außerdem gibt es die Möglichkeit, alle Super- und Kindklassen der schon ausgewählten Klassen hinzuzufügen. Die Super- und Kindklassen werden bei jedem Öffnen des Views erneut ausgewertet, so dass sich die im View enthaltenen Klassen dynamisch ändern können.

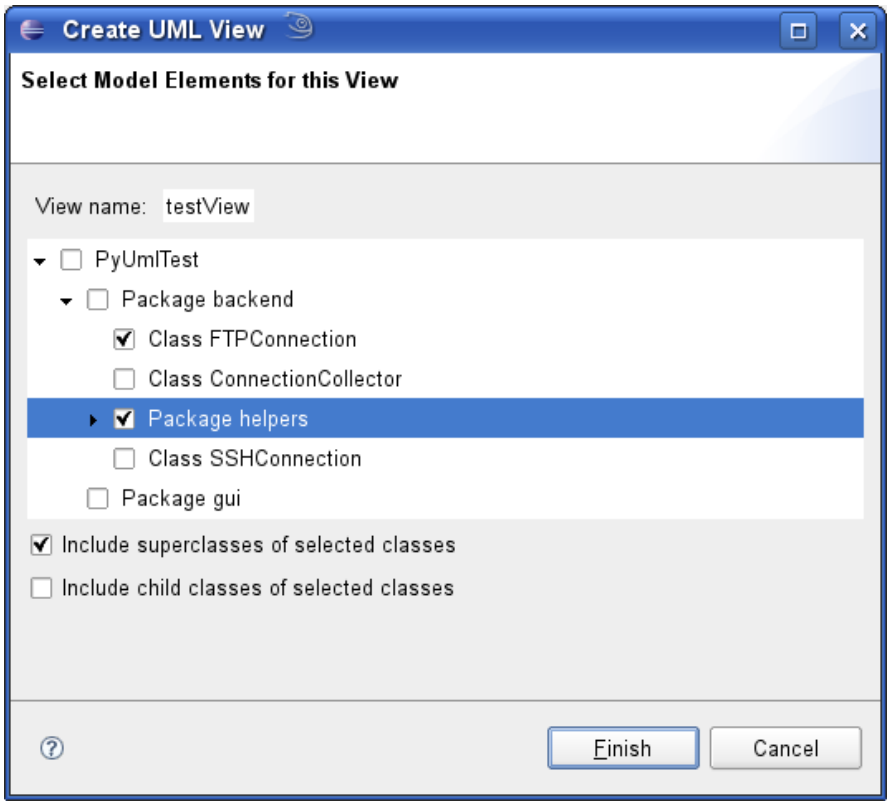

Abbildung 13: Klassenauswahl bei der Erstellung eines Views

Zur Erstellung des Views wird das oben definierte Beispiel verwendet:

Nach dem Beenden durch das Aktivieren von *Finish* wird der View erstellt und automatisch geöffnet.

Er enthält alle ausgewählten Klassen und ihre Verbindungen wie im eigentlichen Modell. Das Interface *Connection* wurde automatisch hinzugefügt, weil *FTPConnection* von *Connection* erbt.

Der View lässt sich nun beliebig bearbeiten, wobei das Hauptdiagramm bei jedem Speichern aktualisiert wird. Allerdings ist zu beachten, dass in den View neu eingefügt Klassen und Pakete nicht übertragen werden können, weil nicht eindeutig definiert ist, unter welchem Paket sie eingefügt werden müssten.

Die Ansicht ist unabhängig vom Hauptdiagramm; es lassen sich beliebige Kommentare hinzufügen oder eine spezielle Anordnung der Elemente vornehmen, die nicht ins Hauptdiagramm übertragen werden. Dadurch kann ein View zur Dokumentation bestimmter Zusammenhänge verwendet werden, ohne andere Diagramme zu beeinflussen. Ein View aus einem realen Projekt kann z.B. wie folgt aussehen:

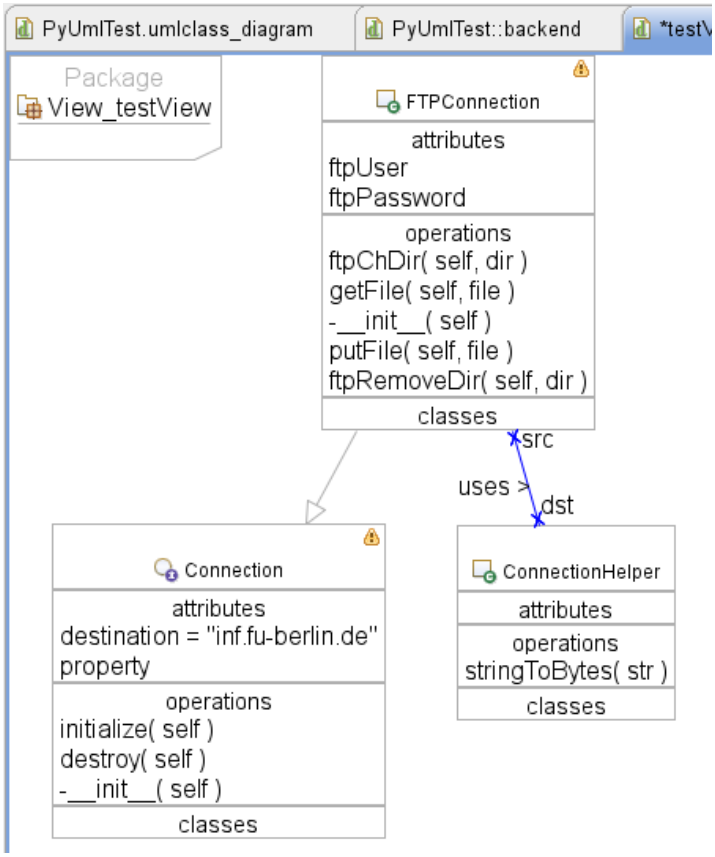

Abbildung 14: Neu erstellter View

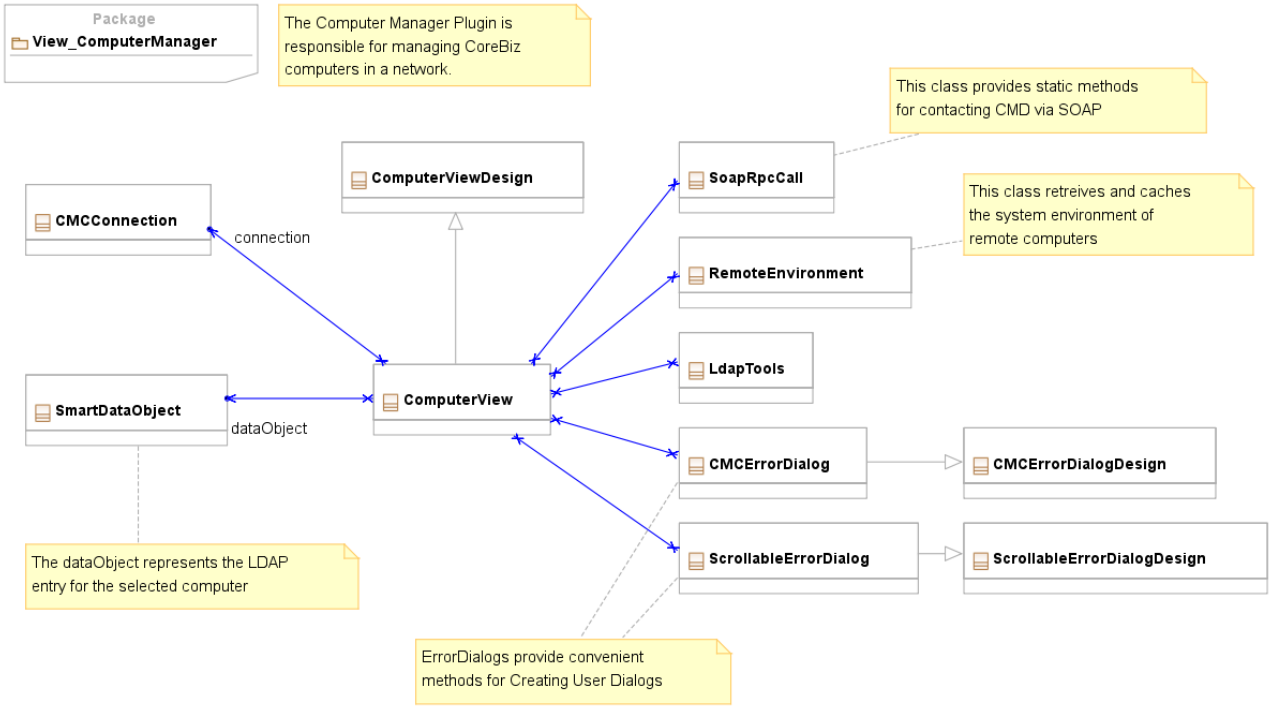

Abbildung 15: Beispiel für einen View (CoreBiz Management Console, Plugin Computermanager)

10 Using the PyUML Profile

TRANSLATION MISSING!

Profile sind Erweiterungen von UML. Sie erlauben die Anwendung so genannter Stereotypen, also zusätzlicher Eigenschaften, die ein Modellelement haben kann. Stereotype können dadurch UML-Elementen eine zusätzliche Semantik für eine bestimmte Domäne, hier Python, geben.

Bei der Erstellung eines PyUML-Modells wird im PyUML-Verzeichnis ein Profil namens PyUML abgelegt. Dieses Profil enthält das Stereotyp *BeanClass*, das auf Klassen angewendet werden kann. Die Verwendung des Stereotyps BeanClass ändert die Codeerzeugung der entsprechenden Klassen in der Weise, dass automatisch für alle Objektattribute *Getter* und *Setter* zum Lesen und Schreiben der Attribute erzeugt werden. Die Erzeugung dieser Getter und Setter funktioniert auch nachträglich bei schon existenten Klassen. Damit erhält eine BeanClass ähnliche Eigenschaften wie eine *JavaBean*.

Um einer Klasse das Stereotyp *BeanClass* zuzuordnen, muss zuerst das PyUML-Profil im Editor aktiviert werden. Dazu wählt man im Kontextmenü eines freien Bereichs im Diagramm die Funktion *Apply Profile→PyUMLProfile*:

Abbildung 16: PyUML-Profil aktivieren

Danach gibt es im Kontextmenü jeder Klasse einen Eintrag *Apply Stereotype→PyUmlProfile::BeanClass*. Über diesen Eintrag kann das Stereotyp gesetzt oder wieder entfernt werden. Gesetzte Stereotype werden in Anführungsstrichen über dem Klassennamen angezeigt (siehe rechts).

Wird danach aus dieser Klasse Code erzeugt, so erhält er automatisch die entsprechenden Getter- und Settermethoden:

Die erzeugten Methoden werden bei der nächsten Modellsynchronisation auch im Modell angezeigt. Wird das Stereotyp im Diagramm wieder abgewählt, bleiben die einmal erzeugten Methoden erhalten.

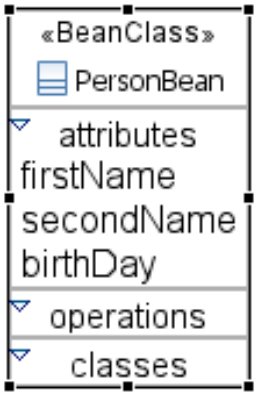

Abbildung 17: Gesetztes Stereotyp BeanClass