DEUTSCHE VERSION

(GERMAN VERSION)

Benutzerhandbuch - User Guide

PyUML for Eclipse

A UML Roundtrip Tool for Python

Version 1.2

Authors: Martin Dittmar

Translators: (none)

Translation Status: (original)

Inhaltsverzeichnis

Dieser User's Guide führt durch die Installation und Benutzung des entwickelten Roundtrip-Tools, hier *PyUML* genannt.

1 Systemvoraussetzungen

PyUML ist ein Plugin für Eclipse und damit auf allen von Eclipse unterstützten Plattformen lauffähig. Eine Liste aller Plattformen findet sich im Download-Bereich von Eclipse[1].

Plattformunabhängige Systemvoraussetzungen sind:

Hardware

- aktuelle CPU, z.B. 2-GHz-CPU
- 512 MB RAM (empfohlen 1 GB)
- ca. 200 MB Speicherplatz auf der Festplatte

Software

- JAVA-Laufzeitumgebung der Version 1.5 oder höher [2]
- Python-Laufzeitumgebung in Version 2.4 oder höher[3]
- grafische Benutzeroberfläche
- beliebiges Eclipse, Version 3.3 oder höher
- PyDev, Version 1.4.2 oder höher (kann bei Installation als Abhängigkeit automatisch installiert werden)

2 Installation

Es gibt zwar mehrere Möglichkeiten, PyUML zu installieren, es wird aber die Installation über den *Eclipse-Update-Manager* empfohlen.

2.1 Benutzen des Update-Managers in Eclipse

Man kann die Eclipse-PyUML-Seite als Softwarequelle (Update Site) eintragen:

Dazu muss man im Menü unter

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

folgende Seites eintragen:

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

und (wenn pyDev nicht schon installiert wurde):

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

Nun wählt man *eclipse-pyUml* aus und klickt auf *Install*. Alle Abhängigkeiten, inkl. *PyDev*, sollten jetzt aufgelöst werden können!

2.2 Installation als Komplettpaket

Hinweis: Ein Komplettpaket gibt es nur für PyUML 1.0 mit Eclipse 3.2. Bitte verwenden Sie stattdessen die Update-Site!

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 als Plugin-Paket

Hinweis: Das Plugin-Paket gibt es nur für PyUML 1.0 mit Eclipse 3.2. Bitte verwenden Sie stattdessen die Update-Site!

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 Einrichten von PyUML

Obwohl PyUML selbst keine Einstellungsseite in Eclipse bietet, sind einige Änderungen der Eclipse-Konfiguration vorzunehmen, damit es reibungslos funktionieren kann.

Ändern der Heapgröße in eclipse.ini

PyUML ist durch den verwendeten Uml2Tools-Editor sehr speicherintensiv. Eclipse beendet sich allerdings beim Überschreiten der erlaubten Speichergröße automatisch. Deshalb empfiehlt es sich, den Eclipse zur Verfügung gestellten Speicher zu erhöhen. Dazu muss die Datei *eclipse.ini* bearbeitet werden.

Die Datei *eclipse.ini* befindet sich im Programmverzeichnis von Eclipse.

Bitte fügen Sie die folgenden 2 Zeilen zu *eclipse.ini* hinzu:

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

Einrichten von PyDev

Bevor Eclipse zur Python-Entwicklung benutzt werden kann, muss das *PyDev*-Plugin eingerichtet werden. Dazu muss PyDev mitgeteilt werden, wo die Python-Laufzeitumgebung zu finden ist.

Um PyDev einzurichten, wählt man in Eclipse unter *Window→Preferences→PyDev* den Punkt *Interpreter-Python*. Nun wählt man im Fenster *Python Interpreters* rechts den Knopf *new* und gibt den Pfad zum Python-Interpreter ein. Unter Linux kann hier meist */usr/bin/python*[4] eingetragen werden, unter Windows liegt Python je nach Installation in einem Unterordner von *C:\Programme*, die auszuwählende Datei heißt hier *python.exe*. Ist der Interpreter gewählt, werden alle verfügbaren Bibliotheken gesucht und die Einrichtung von PyDev ist abgeschlossen. Nun können PyDev-Projekte angelegt und Python-Code ausgeführt werden.

4 Erstellen eines PyDev-Projekts

Da PyUML auf PyDev aufbaut, muss ein PyDev-Projekt existieren, für das ein UML-Diagramm erstellt werden kann. Wenn noch kein PyDev-Projekt vorhanden ist, kann man es unter *File→New→Project->PyDev-Project* erstellen. Der dabei definierte Ordner für Quelldateien wird auch zur Roundtrip-Synchronisation von PyUML verwendet.

Die von PyUML erstellten Dateien, also das UML-Modell, das Modellbackup, das Diagramm und die View-Dateien sowie das PyUML-Profil, werden im Ordner *PyUML* direkt unterhalb des Projektverzeichnisses erstellt. Normalerweise kann man dieses Verzeichnis ignorieren, da alle Funktionen ohne manuelle Auswahl einer PyUML-Datei arbeiten. Dieses Verzeichnis muss allerdings gegebenenfalls in die Versionsverwaltung aufgenommen werden, damit die UML-Dateien für alle Entwickler zugänglich sind.

5 Grundfunktionalität von PyUML

Nach der Installation von PyUML gibt es 4 neue Symbole in der Werkzeugleiste:

Abbildung 1: Plugin-Icons von PyUML

Durch diese Icons lässt sich PyUML steuern. Die Symbole erklären sich der Reihenfolge nach wie folgt:

1. **Modell-Synchronisation**

PyUML sucht in den vom Python-Projekt angegebenen Quellverzeichnissen nach Python-Quellcode sowie nach einem vorhandenen UML-Modell. Wird kein Modell gefunden, wird ein neues Modell anhand des Python-Codes erstellt. In einem leeren Projekt ohne Python-Quellcode kann so ein leeres Modell erstellt werden, das später zur Codeerzeugung genutzt werden kann.

Nach jeder Änderung des Codes muss vor der Bearbeitung des Modells eine Modellsynchronisation durchgeführt werden, damit keine Inkonsistenzen entstehen.

Das erstellte bzw. synchronisierte Modell wird automatisch geöffnet, so dass direkt mit der Bearbeitung begonnen werden kann.

2. **Code-Synchronisation**

Analog zur Modellsynchronisation wird hiermit der vorhandene Python-Code angepasst bzw. erstellt. Nach der Bearbeitung des Modells muss vor Änderungen am Code solch ein Synchronisation stattfinden, damit keine Inkonsistenzen entstehen.

3. **Views-Verwaltung**

Mit dieser Funktion können Views erstellt, geöffnet, bearbeitet und gelöscht werden. Views sind

4 Den tatsächlich verwendeten Python-Interpreter kann man auf der Linux-Kommandozeile durch Eingabe von „*which python*“ erfragen

benutzerdefinierte Ansichten des UML-Modells, die eine Auswahl von Klassen über Paketebenen hinweg enthalten. Sie werden mit dem Hauptmodell synchron gehalten, so dass die Bearbeitung einer Klasse wahlweise im View oder im Hauptmodell stattfinden kann. Eine genauere Erläuterung von Views findet sich weiter unten im User's Guide.

4. **Öffnen eines vorhandenen UML-Modells**

Wenn das UML-Modell schon synchron mit dem Code ist, kann es über diese Funktion ohne eine vorhergehende Synchronisation geöffnet werden. Das ist besonders bei sehr großen Projekten sinnvoll, in denen der Synchronisationslauf eine spürbare Zeit in Anspruch nimmt. Alternativ ist das manuelle Öffnen der Diagrammdatei im *PyUML*-Ordner zwar prinzipiell möglich, allerdings muss das UML-Diagramm mit der von PyUML zur Verfügung gestellten Funktion geöffnet werden, damit Live-Validation angeboten werden kann.

Alle genannten Funktionen suchen zur Ausführung nach PyDev-Projekten im Eclipse-Workspace. Wird nur ein Projekt gefunden, wird es automatisch ohne Rückfrage verwendet. Gibt es mehrere PyDev-Projekte, wird der Benutzer gefragt, welches er bearbeiten möchte. Die letzte Auswahl des Benutzers ist dabei automatisch vorgewählt.

Die Funktionalität der beschriebenen Icons findet sich außerdem im Kontextmenü des Projekts, das mit einem Rechtsklick erreicht werden kann. Durch Verwendung des Kontextmenüs ist das PyDev-Projekt eindeutig vorgegeben, wodurch der beschriebene Auswahldialog nicht angezeigt wird und die Funktionalität gegebenenfalls schneller erreicht werden kann.

6 Benutzung des UML-Editors

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:

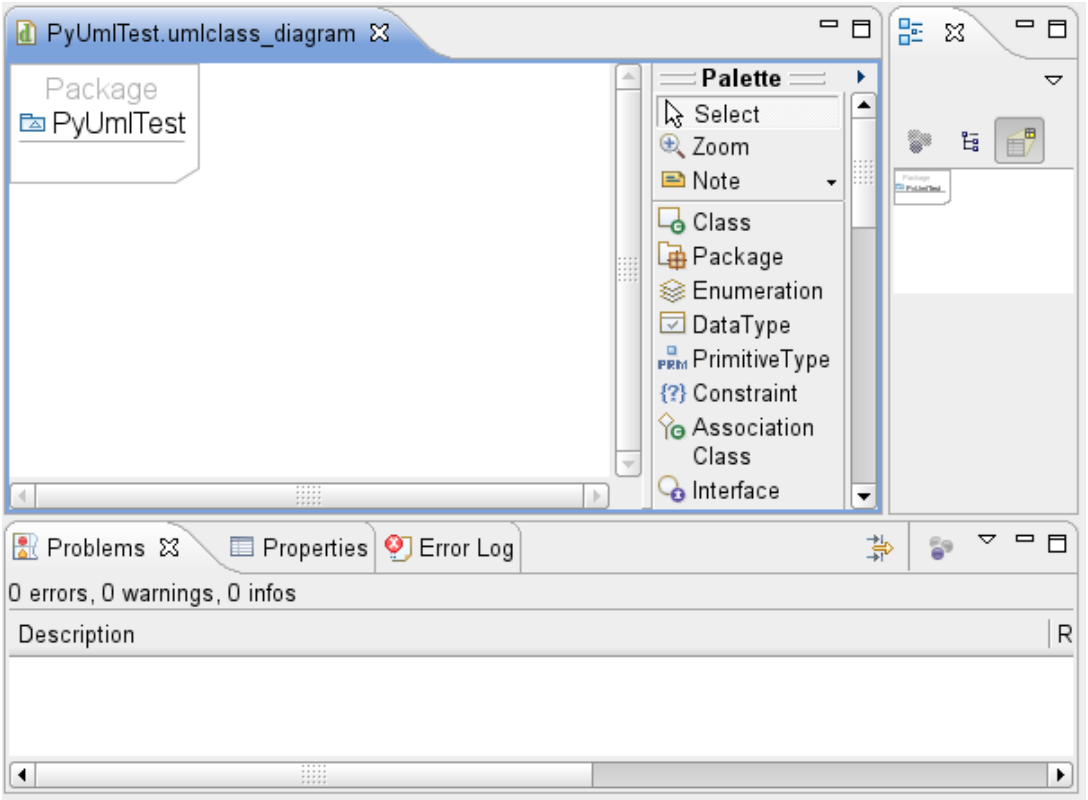

Abbildung 2: Leeres UML-Klassendiagramm

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:

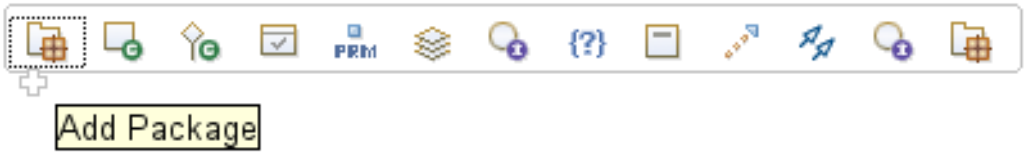

Abbildung 3: Kontextabhängige Palette im UML-Diagramm

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:

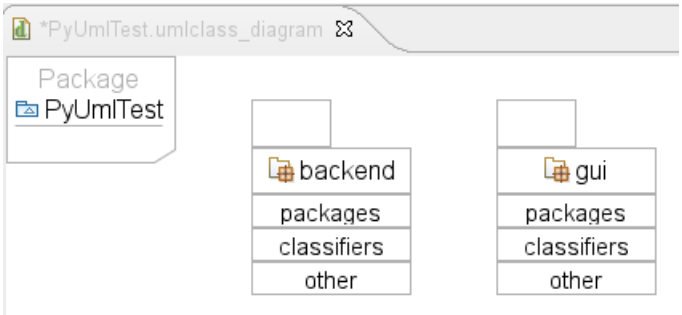

Abbildung 4: UML-Editor: Erstellung von Paketen

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:

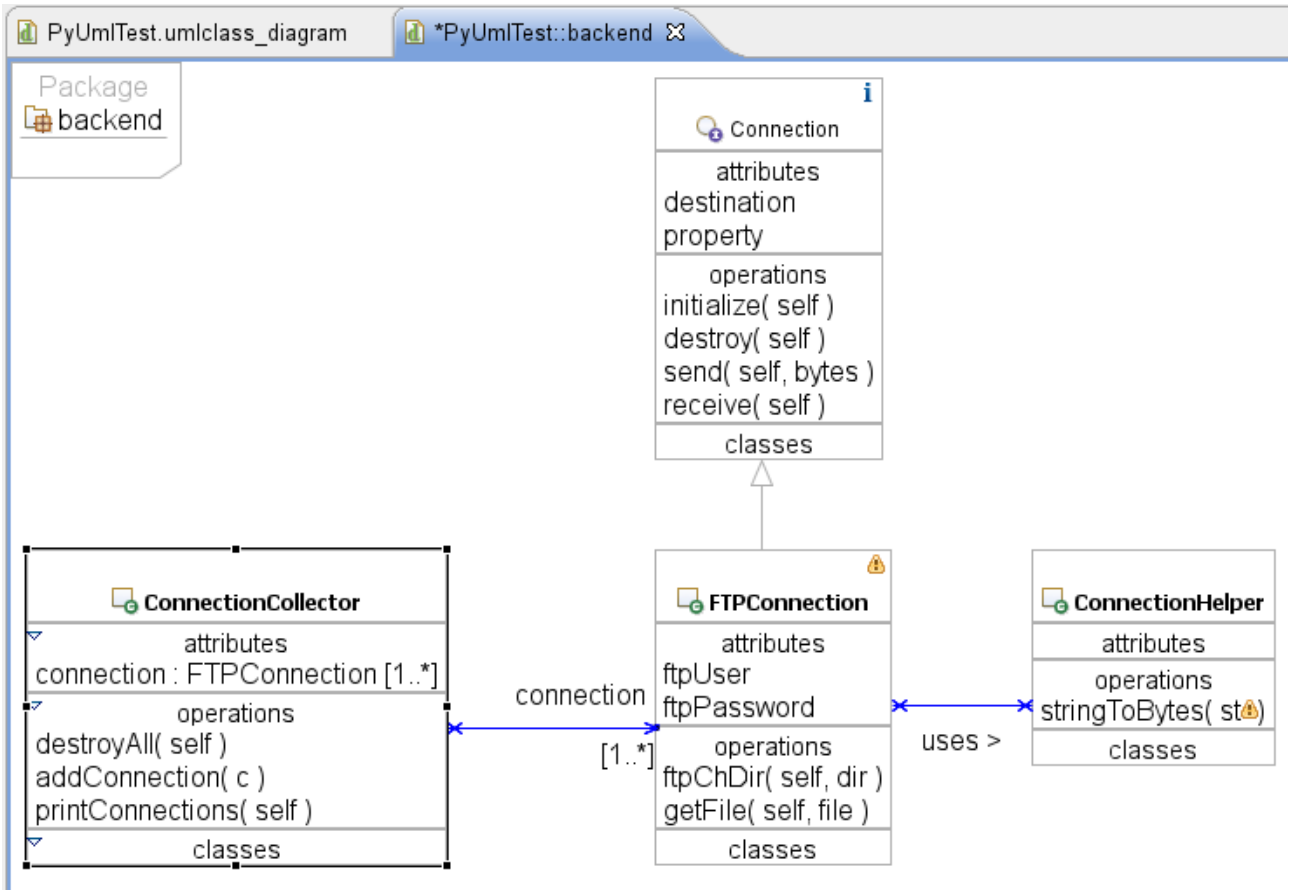

Abbildung 5: Beispielklassen im Paket backend

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.

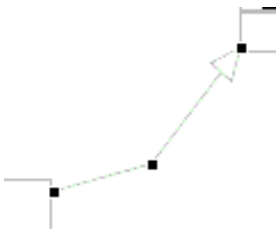

Abbildung 6: Einfügen eines Eckpunktes

7 Roundtrip-Funktionalität

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.

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.

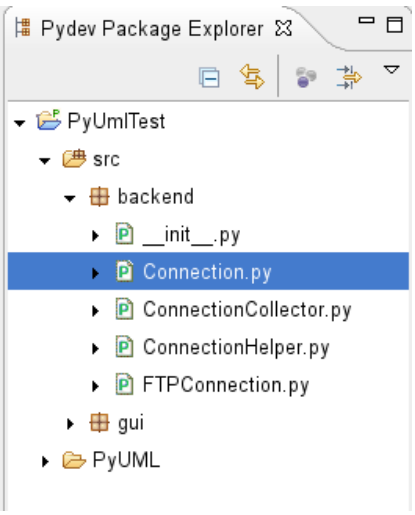

Abbildung 7: Automatisch erstellte Pakete und Klassen

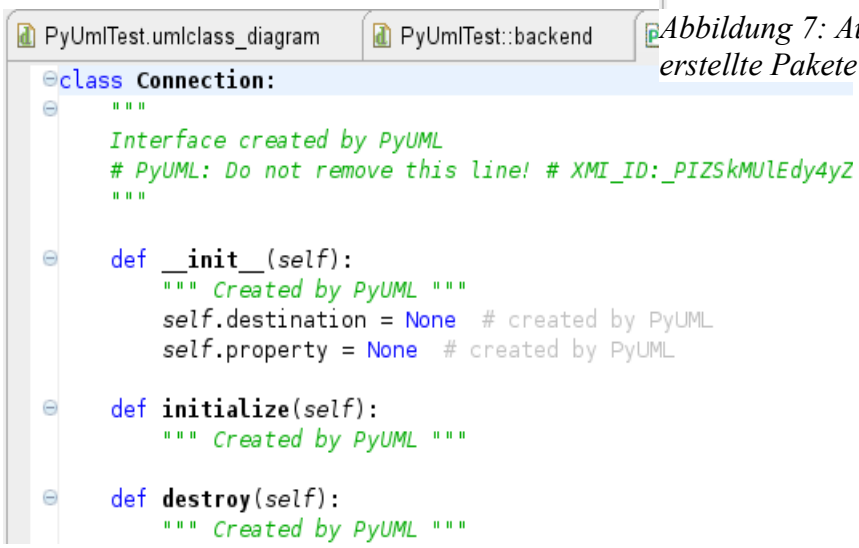

Abbildung 8: Automatisch aus Interface erstellte Klasse Connection

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

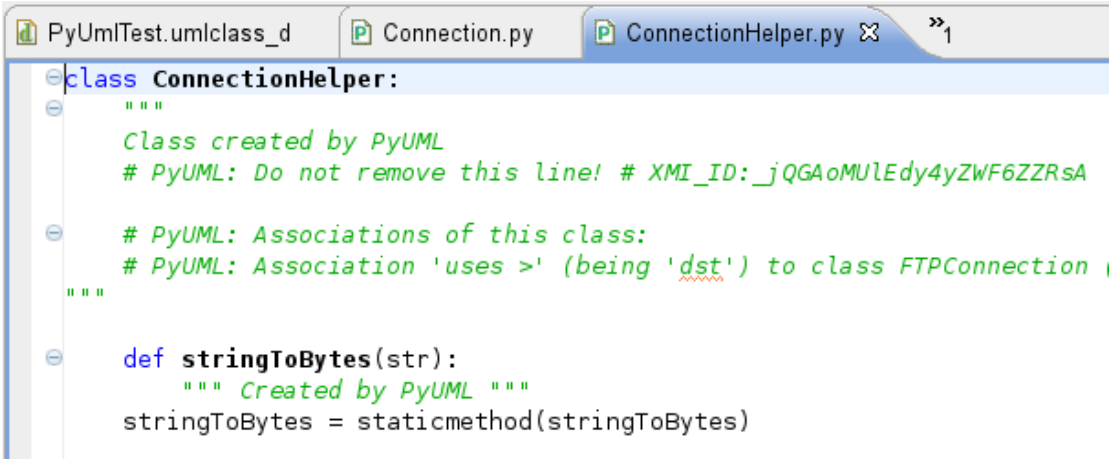

Abbildung 9: Automatisch erstellte Klasse ConnectionHelper mit Darstellung der Assoziationen und mit statischer Methode

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:

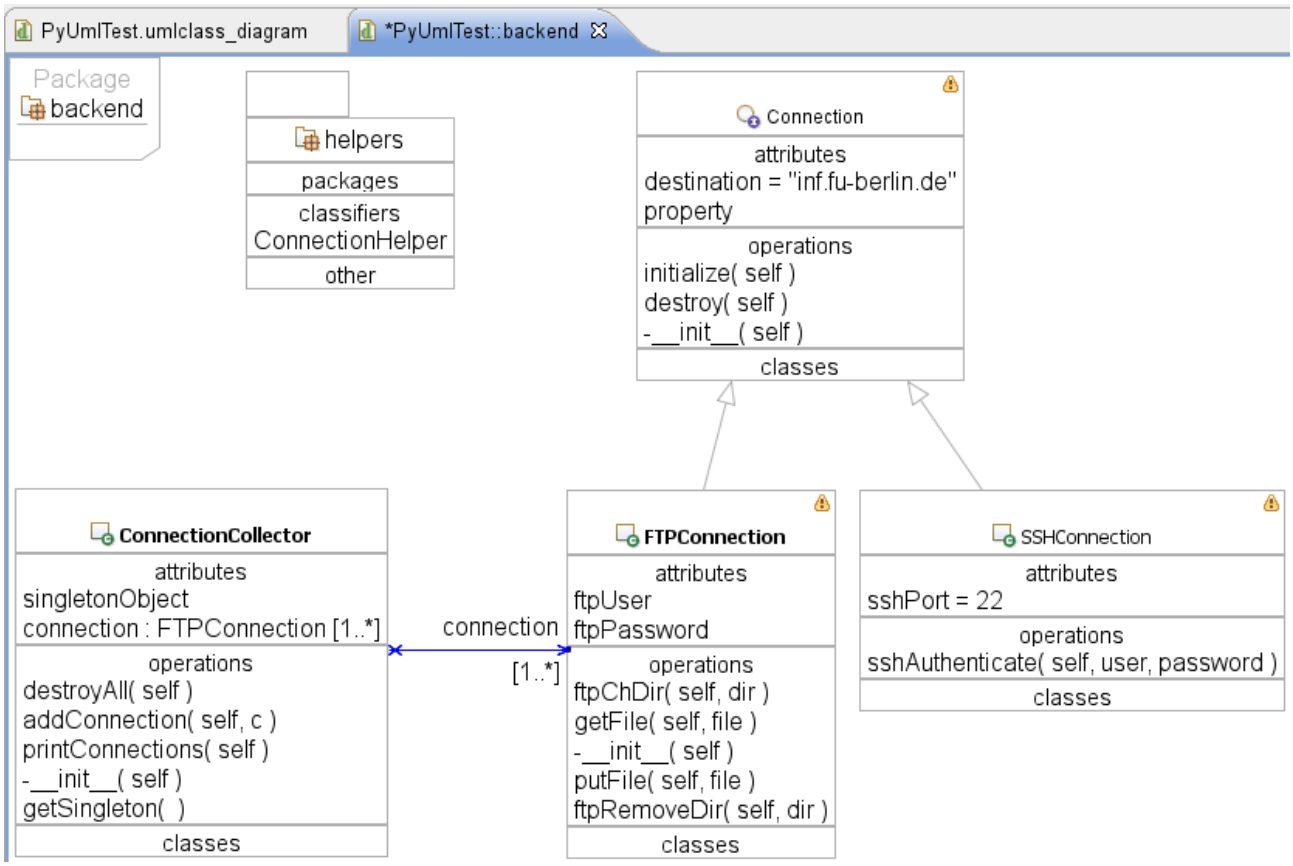

Abbildung 10: Modell nach Synchronisation

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

Bitte beachten Sie, dass Live-Validation nur in PyUML 1.0 funktioniert. In der aktuellen Version wurde Live-Validation abgeschaltet, weil hierfür der jeweils aktuelle Uml2Tools-Editor bearbeitet werden müsste!

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.

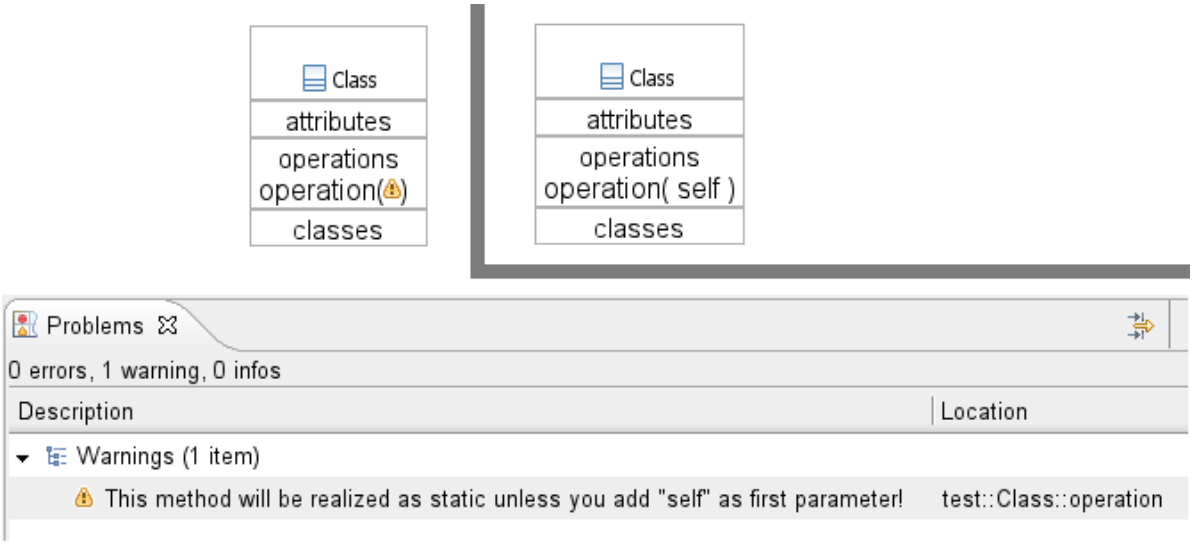

Abbildung 11: Live-Validation: Klasse mit angezeigter Warnung und zugehörigem Problems View und Klasse mit verschwundener Warnung

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.

9 Views

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:

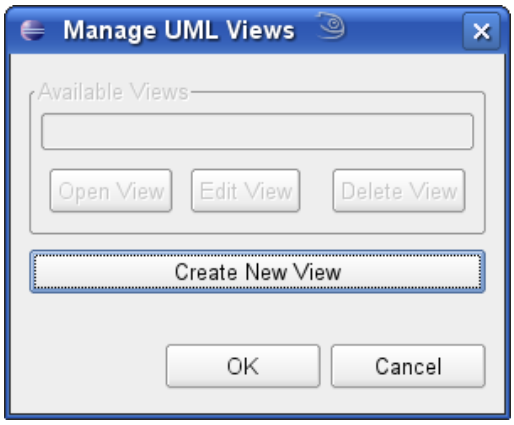

Abbildung 12: Initiale Views-Verwaltung

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.

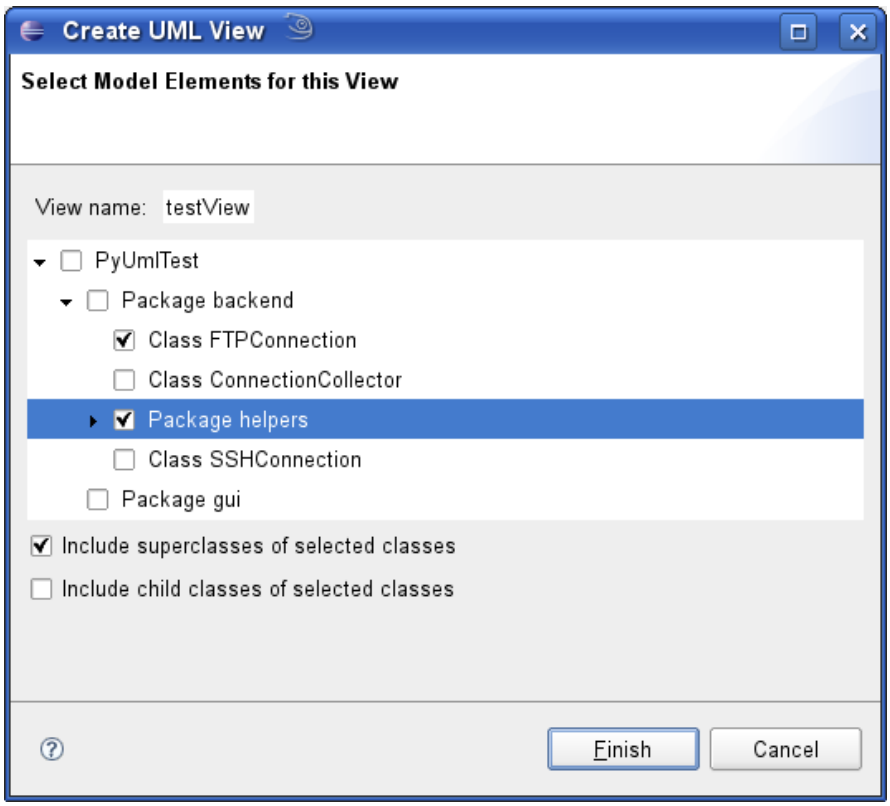

Abbildung 13: Klassenauswahl bei der Erstellung eines Views

Zur Erstellung des Views wird das oben definierte Beispiel verwendet:

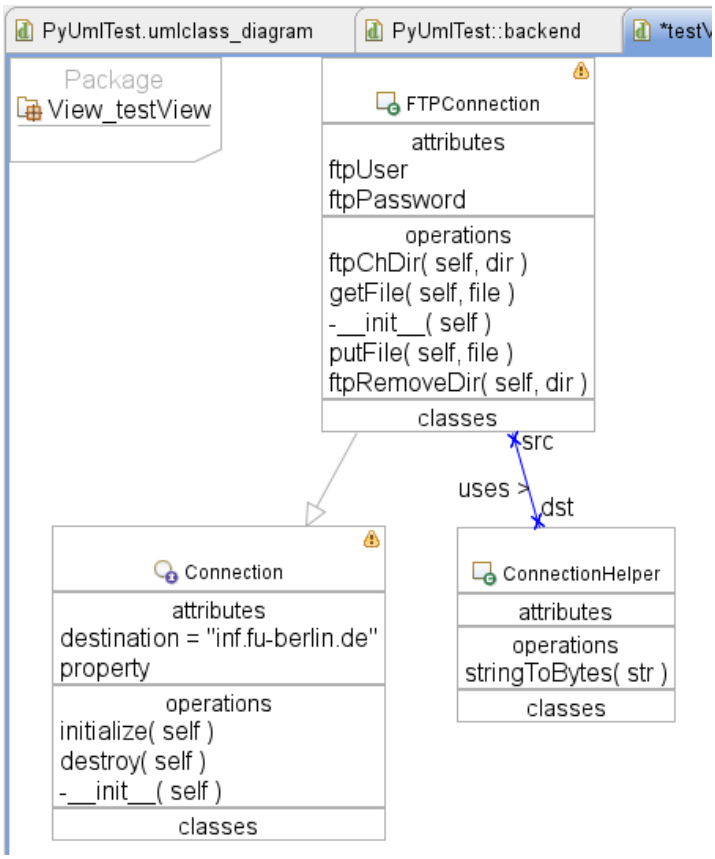

Abbildung 14: Neu erstellter View

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:

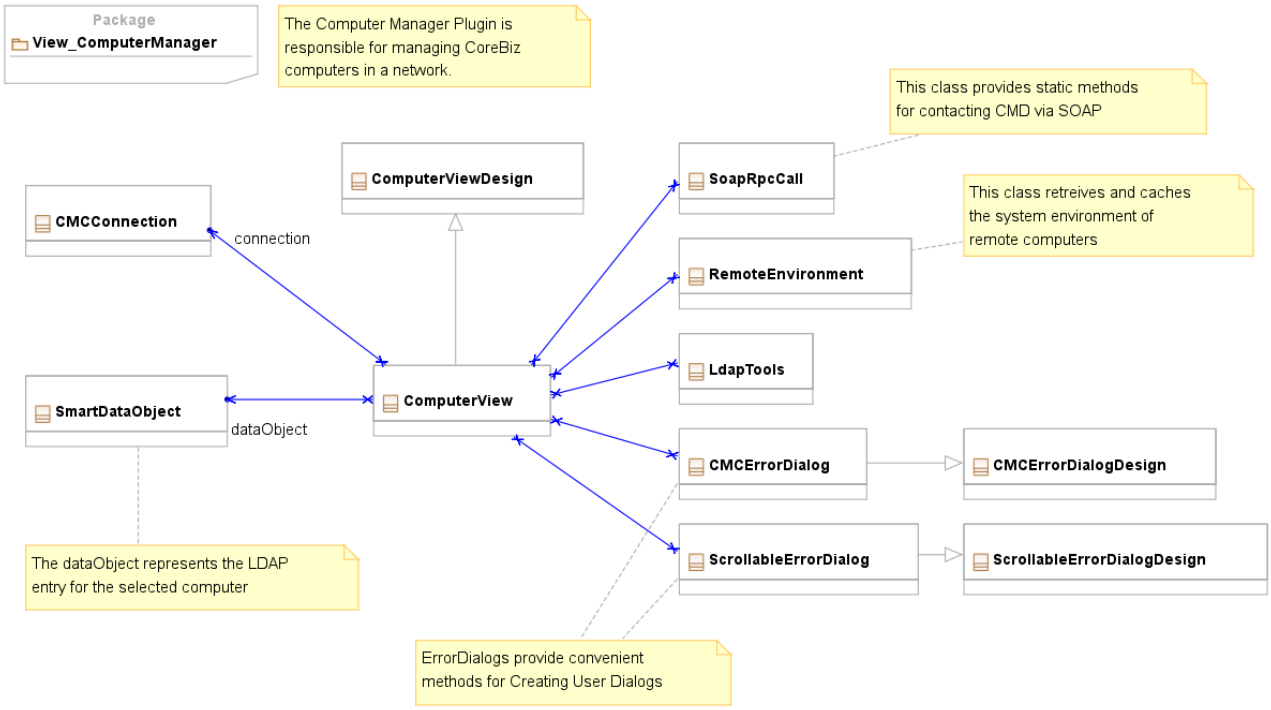

Abbildung 15: Beispiel für einen View (CoreBiz Management Console, Plugin Computermanager)

10 Einbindung des PyUML-Profils

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).

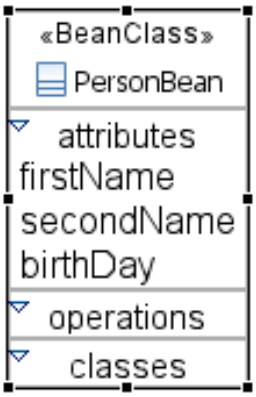

Abbildung 17: Gesetztes Stereotyp BeanClass

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.