

UML4CPP

Tutorial

For Windows

Table of Contents

1	Introduction.....	3
1.1	Motivation and Aim	3
1.2	Entry point of this Guide.....	3
1.3	Important Notes	3
2	Creating an UML Diagram and deriving Code using MDE4CPP.....	4
2.1	Starting eclipse via Command Line.....	4
2.2	Creating a new Project.....	5
2.3	Initializing an UML diagram	10
2.4	Creating a basic class diagram using UMLDesigner.....	12
2.4.1	Creating a class and adding attributes.....	12
2.4.2	Adding generalizations and associations	17
2.4.3	Adding functions.....	21
2.4.4	Implementing function behavior.....	25
2.4.5	Creating a main function	32
2.5	Generating and compiling the model	41
3	Running the application	42
4	Outlook and in-depth literature	44
4.1	Activity diagrams.....	44
4.2	fUML	44
4.3	PSCS	44
4.4	OCL.....	44

1 Introduction

These instructions are intended to give an insight into the use of MDE4CPP (Model Driven Engineering for C++) and to show the possibilities it offers when generating code. The following examples are created and tested only on a Windows system. MDE4CPP does support both Windows and Unix systems, but the examples may only work on Unix systems if small changes are made. It is also useful, but not necessary, to have some knowledge of C++. This document follows a reference model, shown in Figure 1.

1.1 Motivation and Aim

MDE4CPP is a useful tool that makes the step from model to source code much easier by deriving code from given UML diagrams, which saves a lot of time and effort. Another advantage is that the automatic code generation means that fewer errors flow into the software during development, so that the code quality is better. The aim is for the user to be able to independently generate code from UML diagrams using MDE4CPP after processing these examples.

1.2 Entry point of this Guide

A basic requirement is that MDE4CPP has been successfully installed and is fully functional. This means, you should have a working eclipse installation (containing the UML-Designer), a working compiler (MinGW), CMake and of course MDE4CPP. If this is not the case, a comprehensive guide can be found [here](#). It is also very helpful to have already worked through the “Ecore for MDE4CPP Tutorial”, which can be found here: [Ecore for MDE4CPP Tutorial](#). This document leads the user through the whole process of creating a UML diagram and deriving code via MDE4CPP, while explaining and recreating the reference model shown in Figure 1.

1.3 Important Notes

For the given examples, it is highly recommended to use the same names for parameters and files as well as the same directory paths shown in this document. On the one hand, this should prevent some common mistakes like using own parameter names and mixing them with the ones from the examples. On the other hand, some paths and identifiers must have certain names, because they are relevant for the code generator and the compiler.

If some steps are incomprehensible or do not work for any reason, the complete and functioning example can be found at [the official MDE4CPP website](#).

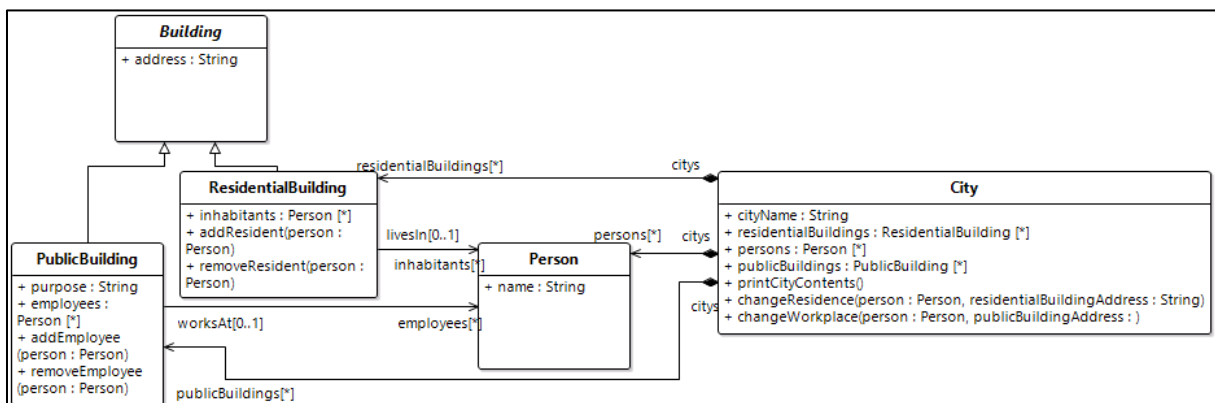


Figure 1: Reference model

2 Creating an UML Diagram and deriving Code using MDE4CPP

2.1 Starting eclipse via Command Line

At first, it is necessary to open the Command Line Window and navigate to the MDE4CPP installation folder. This can be achieved by pressing the Windows key and the R key at the same time (then the Run window opens), entering “cmd”, then clicking “OK” and entering “cd <path-to-MDE4CPP>”, where <path-to-MDE4CPP> is the path to the installation directory. This sets the workspace of the command line to the MDE4CPP home directory. Another way is to open Windows Explorer, navigating to the MDE4CPP installation directory, clicking on the directory path (marked in Figure 2), and typing “cmd”, then pressing Enter. This will also open the Command Line Window in the installation directory.

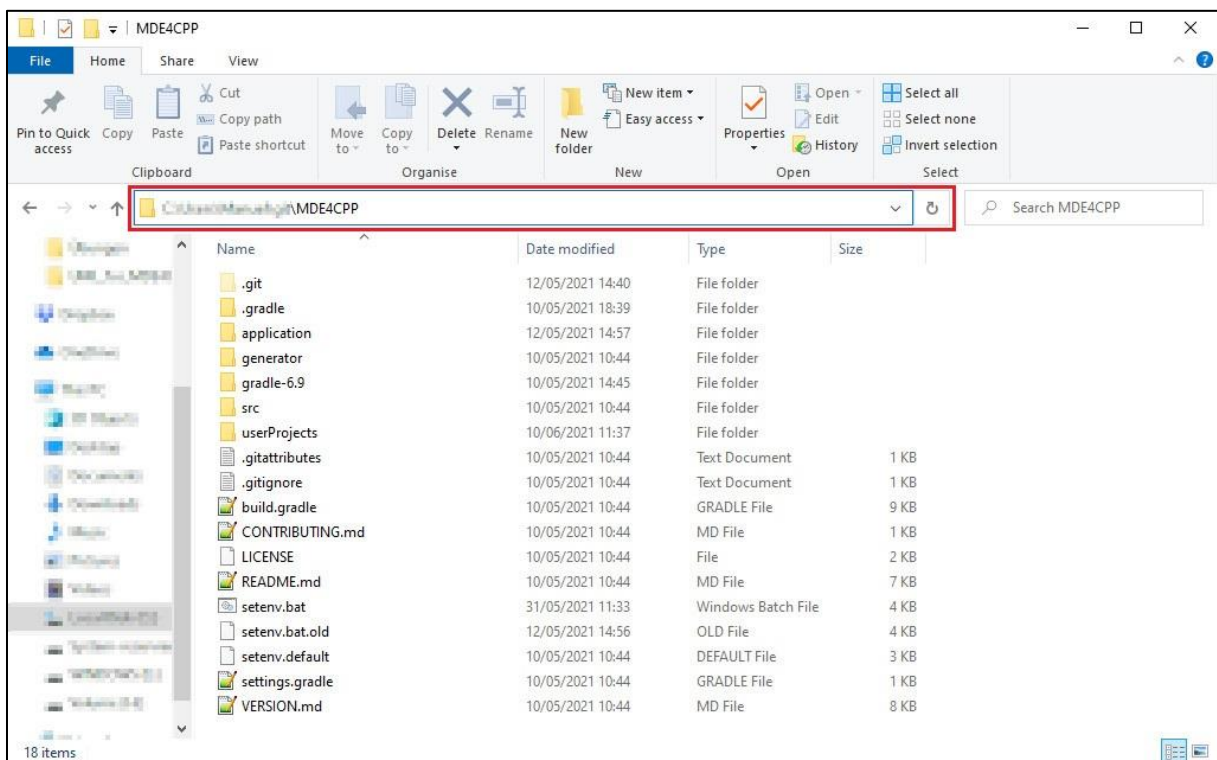
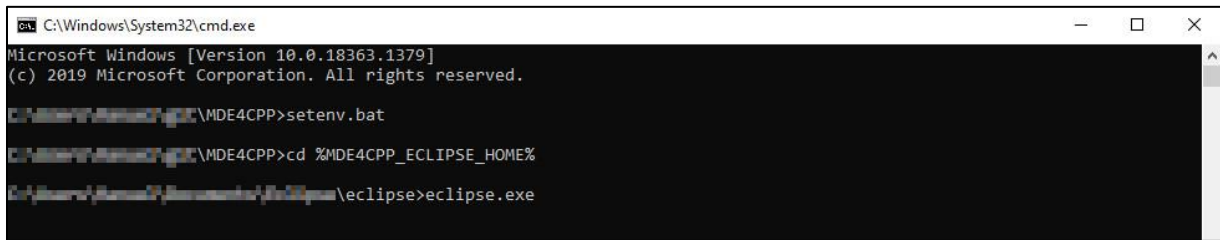


Figure 2: MDE4CPP installation directory

At first, it is necessary to set all required environment variables by typing “setenv.bat” into the Command Line Window and pressing Enter. This file should already have been configured during the installation of MDE4CPP. The next step is to navigate the Command Line to the eclipse installation directory. The easiest way is to type “cd %MDE4CPP_ECLIPSE_HOME%”. Alternatively, the path of the eclipse installation directory can also be copied and pasted manually. After that, typing “eclipse.exe” (as shown in Figure 3) and pressing Enter will start eclipse with all mandatory environment variables already set.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18363.1379]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\MDE4CPP>setenv .bat
C:\MDE4CPP>cd %MDE4CPP_ECLIPSE_HOME%
C:\MDE4CPP\Eclipse\workspace\workspace>eclipse>eclipse.exe
```

Figure 3: Command line after navigating to the eclipse directory and typing “eclipse.exe”

After that, a window will pop up asking for the workspace path. It is strongly recommended to choose a location outside of the MDE4CPP directory for the workspace. An example is shown in Figure 4. In addition, a checkmark can be set by clicking “Use this as the default and do not ask again” so that the window will not open again the next time eclipse is started. Clicking “Launch” will open eclipse.

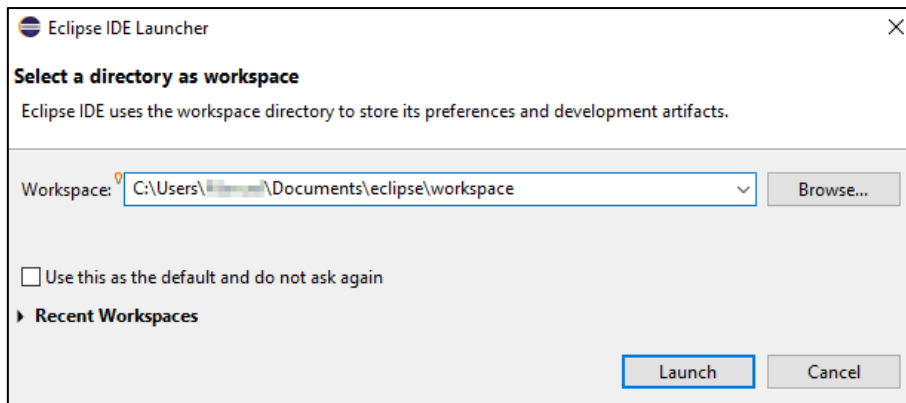


Figure 4: eclipse launcher with workspace settings

2.2 Creating a new Project

When eclipse has opened, the first step to create an UML project using MDE4CPP is to import MDE4CPP into the eclipse workspace. Right-click in the Model Explorer to the left and click “Import...” in the new menu, as shown in Figure 5. Select “Existing Projects into Workspace” in the “General” section as can be seen in Figure 6, then click next. Click “Select root directory” and type in the path to the location of your MDE4CPP installation or click “Browse...” and select the root directory of your MDE4CPP installation. After that, the window should look like Figure 7, then click “Finish” to import MDE4CPP.

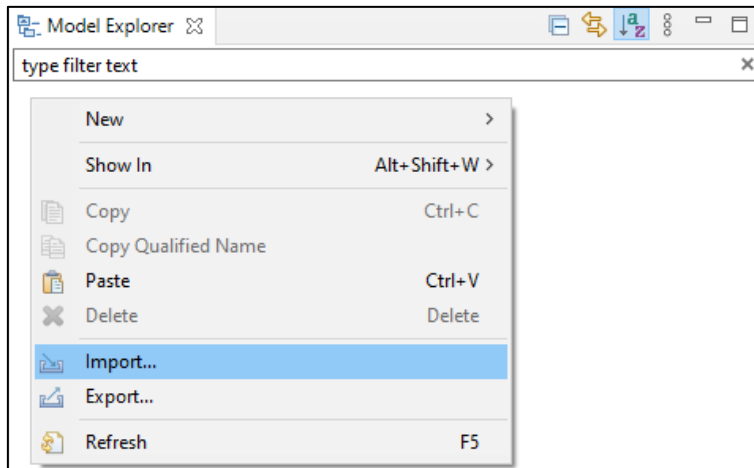


Figure 5: Importing MDE4CPP into the eclipse workspace (1)

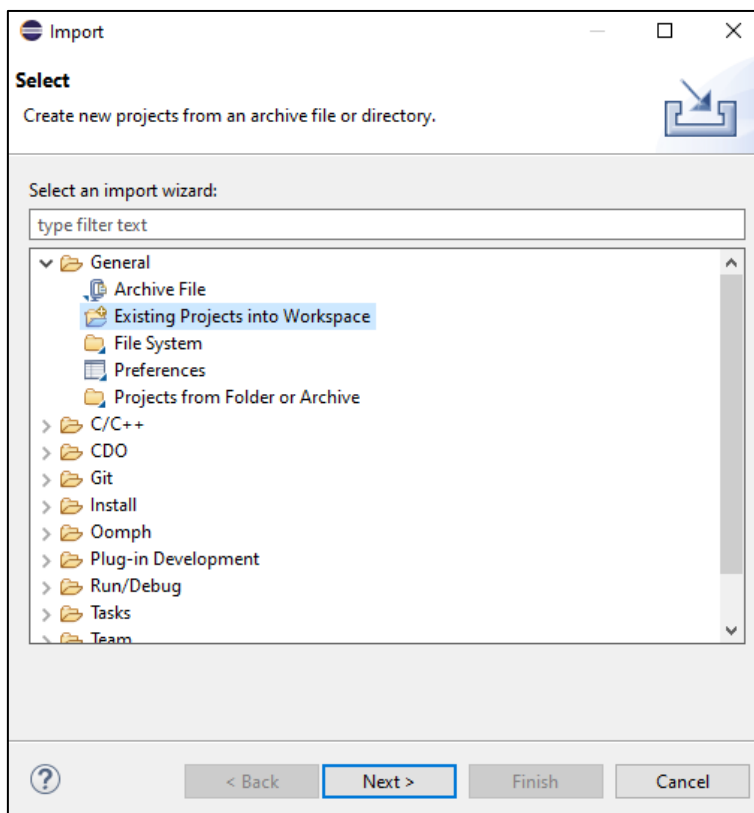


Figure 6: Importing MDE4CPP into the eclipse workspace (2)

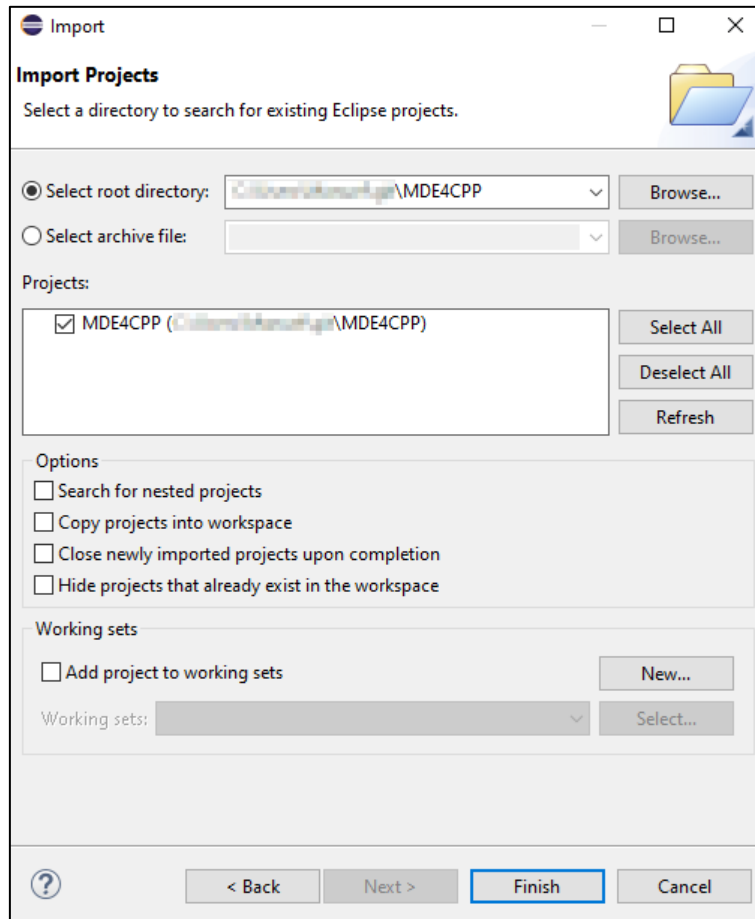


Figure 7: Importing MDE4CPP into the eclipse workspace (3)

A new project can be created by right clicking the “userProjects” folder in the model explorer as shown in Figure 8. At “New”, clicking the option “Others...” will open a new window displaying all available wizards. Search for a project template called “UML Project” within the category “UML Designer” by typing “UML Project” into the search bar (as shown in Figure 9), click on it and click “Next”.

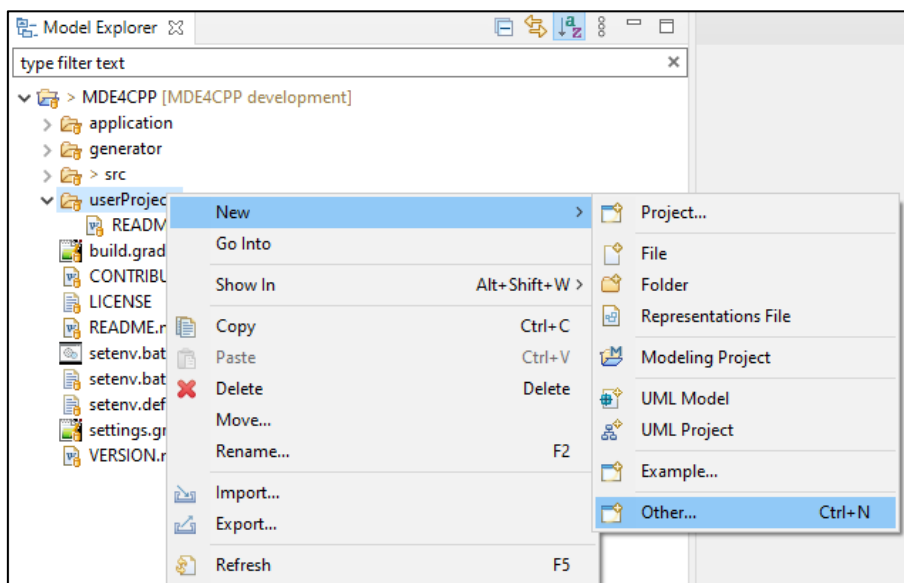


Figure 8: Creating a project inside the userProjects directory

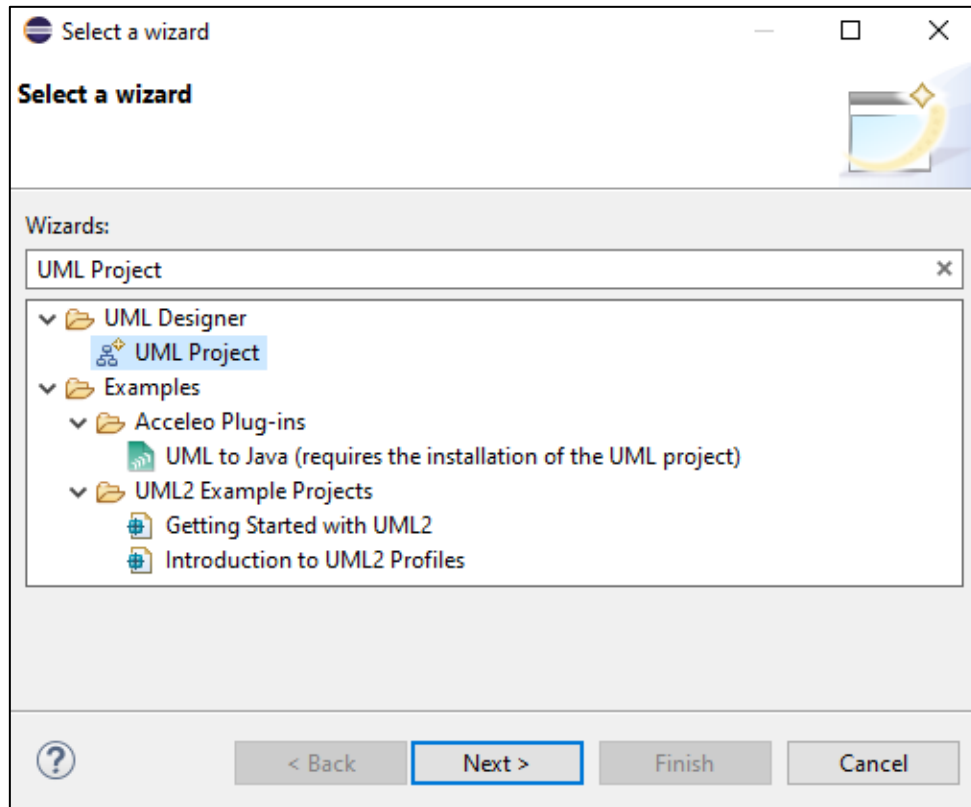


Figure 9: Selecting a project wizard

After pressing next, type in a project name. In this example, the project will be named “UML_Example” as it can be seen in Figure 10. Create a folder inside the userProjects folder, name it “UML_Example” and choose the new folder as project location. Then click “Next” and on the next page, select “Model” as Model Object (see Figure 11) and click Finish. It might take some seconds until the project is fully prepared. The project then should be shown in the Model Explorer on the left inside the “userProjects” directory consisting of two items (model.uml and representations.aird) as shown in Figure 12.

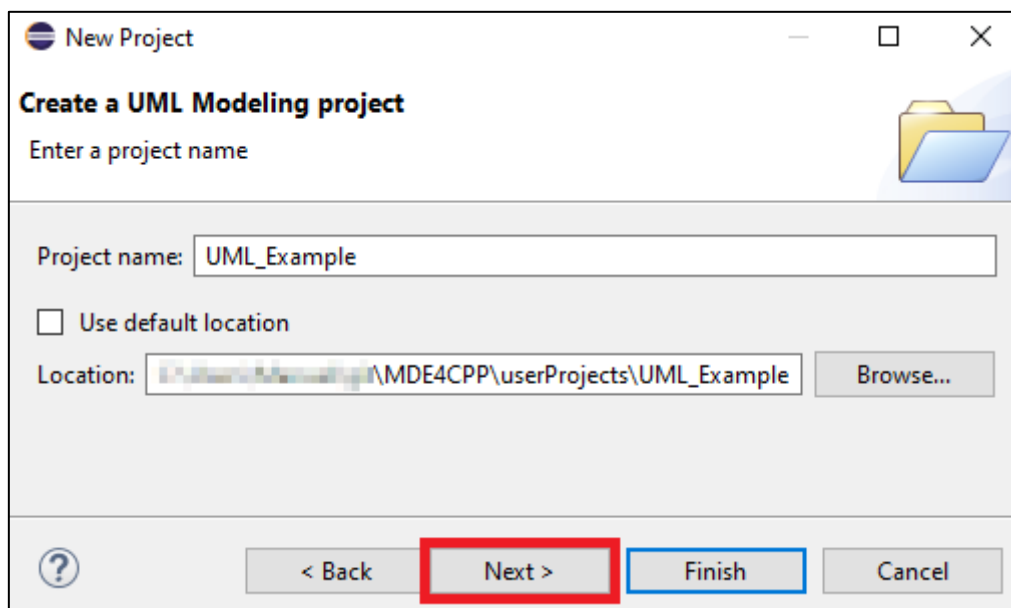


Figure 10: choosing project name

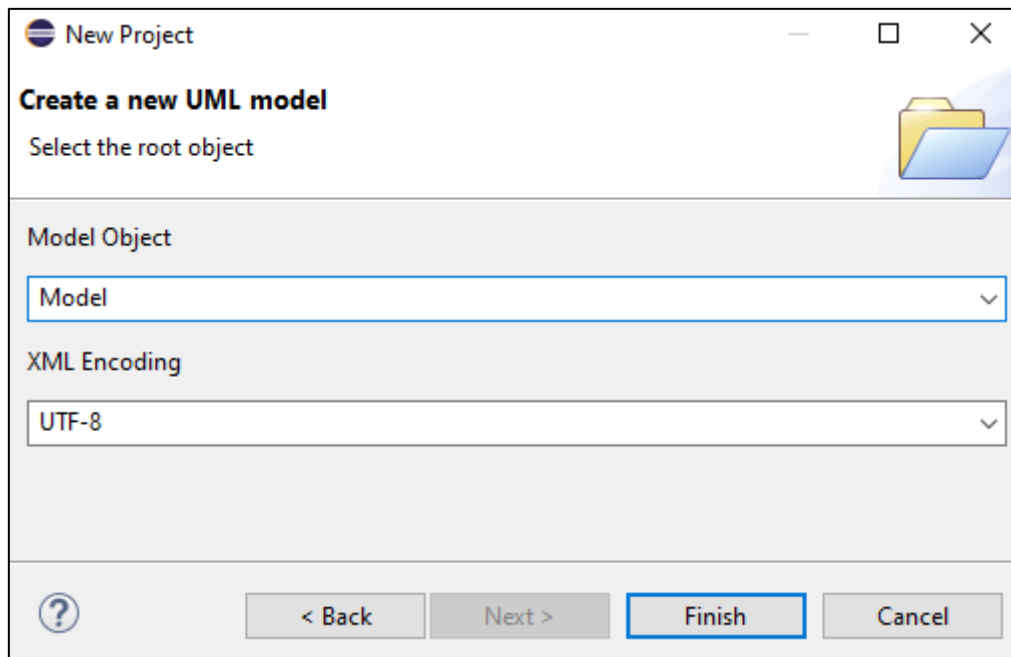


Figure 11: Choosing the model object

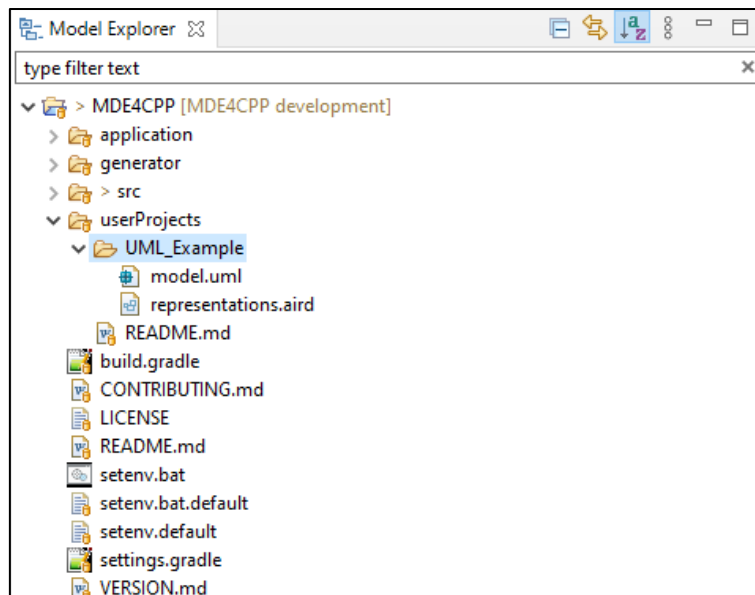


Figure 12: Initial project structure

The “model.uml” has to be in a directory called “model”. This is required by the MDE4CPP generator. Right-click on “UML_Example”, click on “New” and select “Folder”. Name the folder “model” and click “Finish”. Then right-click on “model.uml” and click on “Move...”. Select the “model” folder and click “OK”. Then change the name of the “model.uml” by right-clicking on “model.uml” in the model explorer, clicking “Rename” and entering “CityModel.uml” as the name for the example model. The result can be seen in Figure 13. Notice that the “CityModel.uml” file is indented more than “representations.aird”, which shows that the UML file is in fact inside the model folder. Also, delete the “representations.aird” file. A new representation file will be created in chapter Initializing an UML diagram. The deletion is important since multiple existing representation files might cause errors.

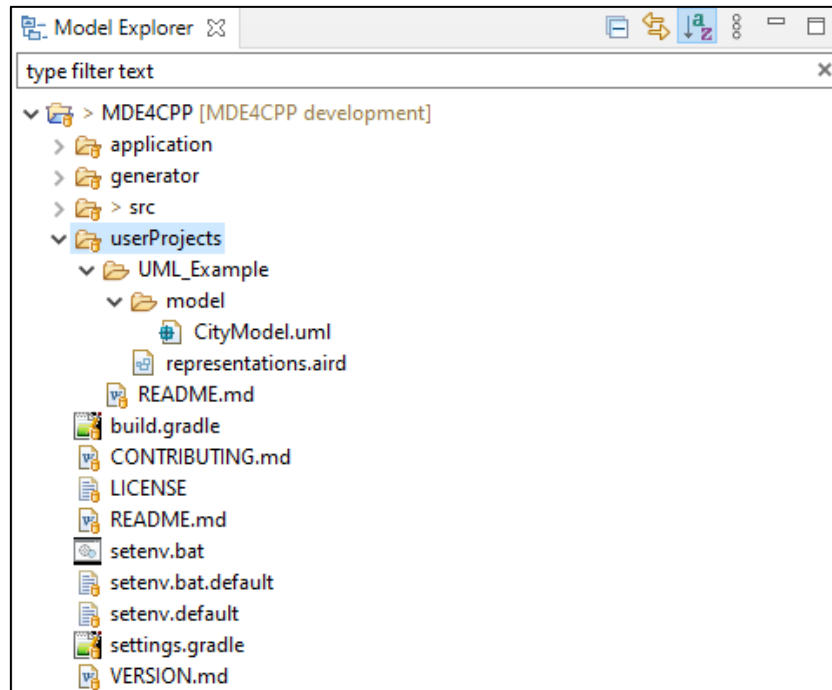


Figure 13: Model Explorer with modified name and project structure

2.3 Initializing an UML diagram

When eclipse has set everything, right-click on the file “CityModel.uml” in the Model Explorer to the left. Click on “Initialize UML diagram ...” in the new menu as shown in Figure 14. A new window will open asking for the name of the new representation file, as can be seen in Figure 15. Type in “CityModel.aird” if it is not already selected. Also, make sure the model folder is selected. Then click Finish.

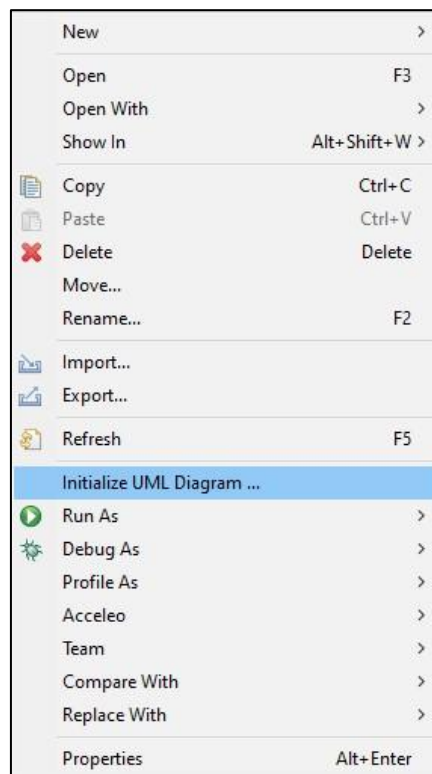


Figure 14: CityModel.uml menu

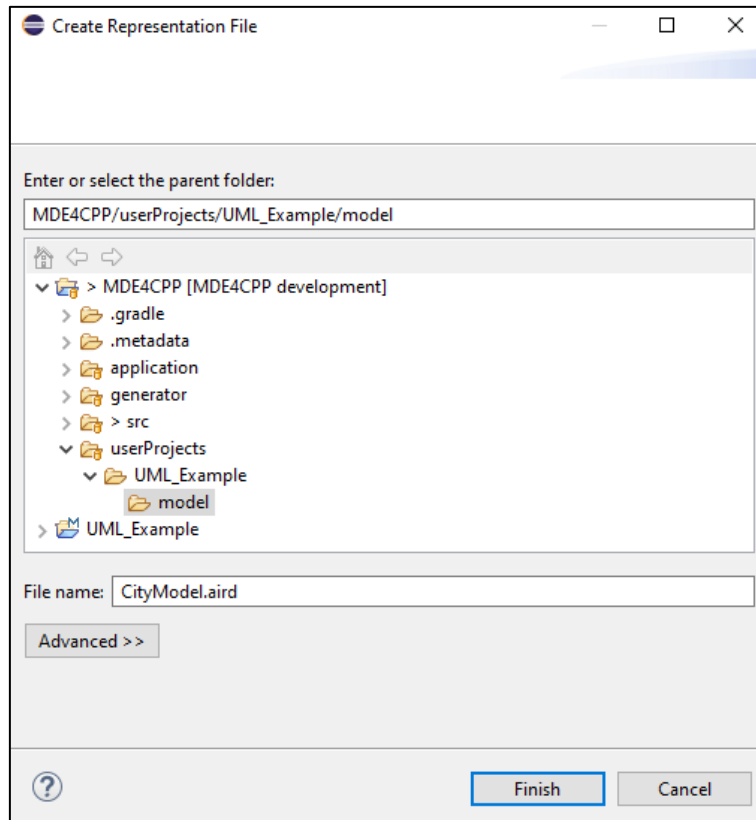


Figure 15: Creating a new representation file inside the model folder

Again, a new window will open asking for a representation type. Choose “Class Diagram” from the “Design” section, click “Next” and then “Finish” (see Figure 16).

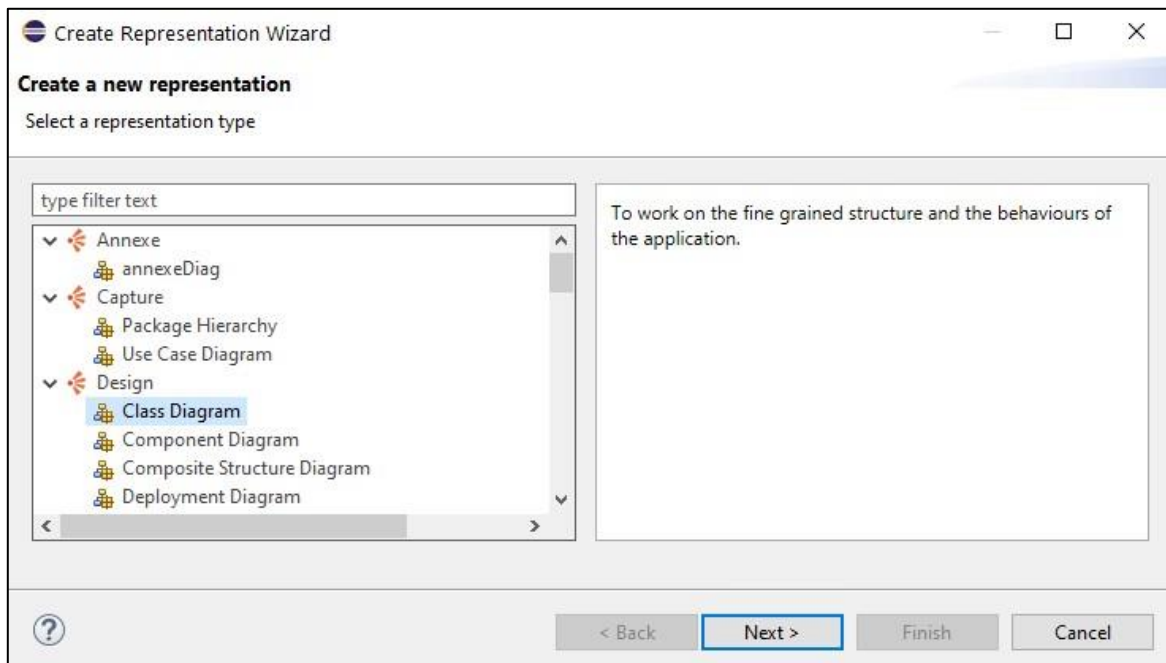


Figure 16: Creating new representation

After that, choose a name for the class diagram. Here it is “CityModel Class Diagram” as shown in Figure 17. Then press OK and wait until the UMLDesigner has opened.

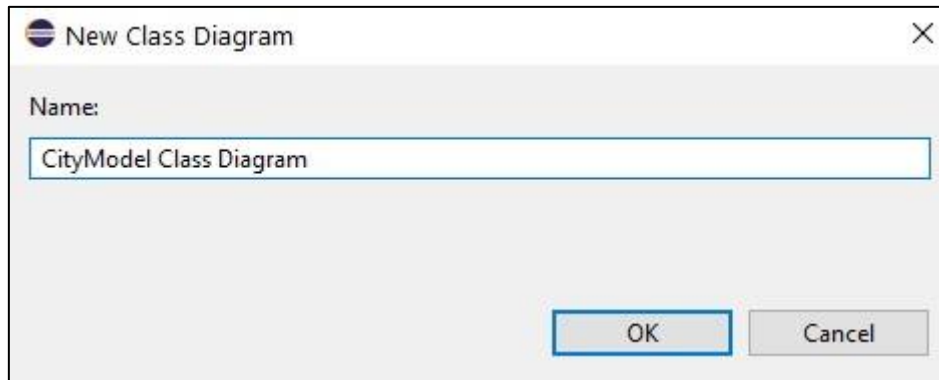


Figure 17: Entering diagram name

2.4 Creating a basic class diagram using UMLDesigner

2.4.1 Creating a class and adding attributes

When creating class diagrams, it is important to understand the basics of the Palette of the UMLDesigner. The Palette, highlighted in the workspace that is shown in Figure 18, is structured in the four categories “Existing Elements”, “Types”, “Features” and “Relationships”. Only some of the basic functions are explained here, where most of them are needed to create the example model:

- By clicking “Add” in the category “Existing Elements” and clicking somewhere in the class diagram, you can import existing class diagrams.
- Clicking “Class” in the category “Types” and clicking somewhere in the class diagram, a new window opens, where you can type the name of the new class and set the visibility. By clicking OK, the new class is created at the location, where you first clicked.
- Clicking “Property” in the category “Features” and clicking on an existing class in the diagram opens a new window, where you can set the name of the new attribute as well as visibility, type and more. Clicking OK will close the window and create a new attribute in the selected class.
- Clicking “Operation” in the category “Features” and clicking on an existing class in the diagram also opens a new window, where you can set the name of the new method, visibility and more. Clicking OK will close the window and create a new method in the selected class.
- Clicking “Association” in the category “Relationships” and clicking on two existing classes in the diagram creates a connection between these classes. This will be further explained later. Clicking on the small arrow to the left of the word “Association” in the category “Relationships” shows a new menu with similar options like Composition and Aggregation, which will also be further explained when using it in the example model.
- Clicking “Generalization” in the category “Relationships” and clicking on two existing classes in the diagram creates a generalization between these classes, where the first-clicked class is the more specific class, and the second-clicked class is the general class.

Most placed objects in the UMLDesigner workspace like classes and associations can be moved via drag and drop to provide a better overview.

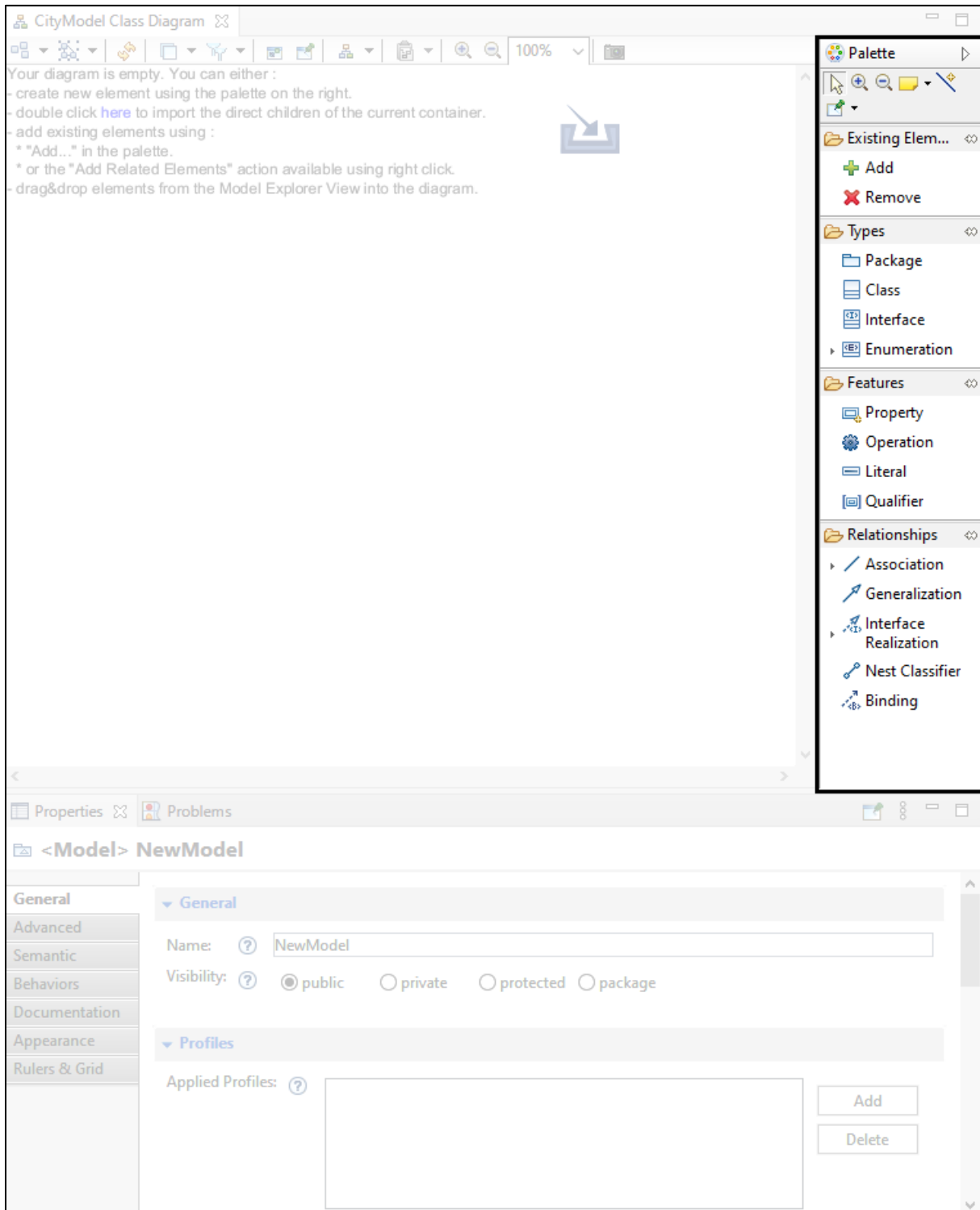


Figure 18: UMLDesigner workspace, Palette on the right is highlighted

The first step to recreate the example model is to add basic classes to the model. Add four classes with the names “City”, “Person”, “PublicBuilding” and “ResidentialBuilding” as described above. Add a fifth class named “Building” and activate the field “Abstract” by ticking the box to the left. The class diagram should then look like the one shown in Figure 19. Notice that the name of the “Building” class is italicized indicating that it in fact is abstract.

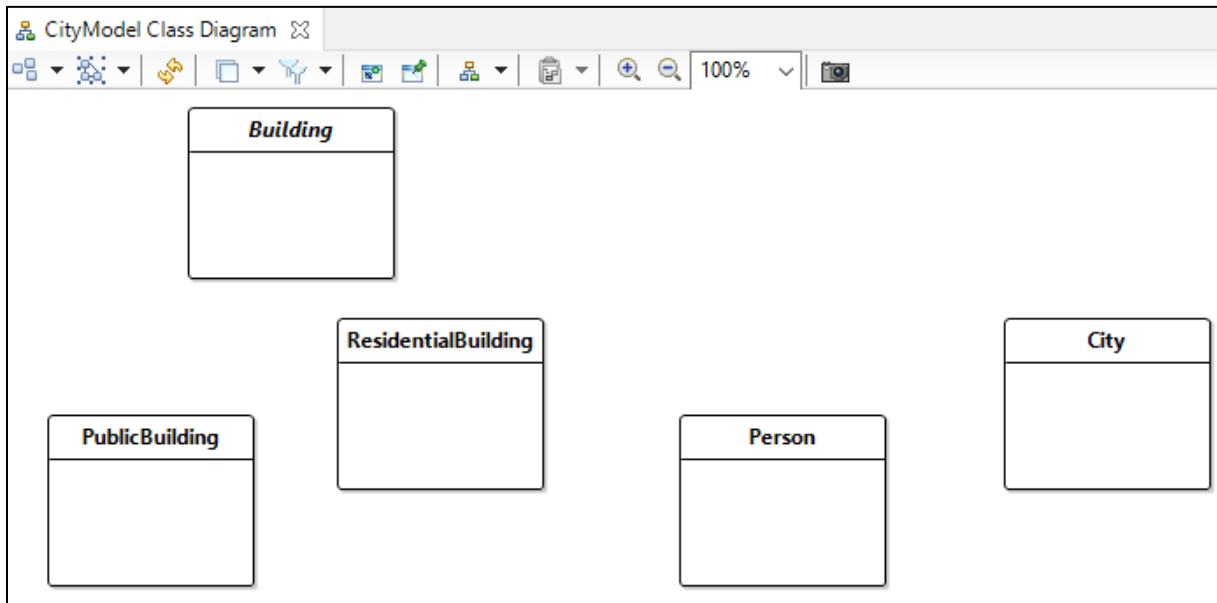


Figure 19: class diagram after creating the five basic classes

The next step is to add some basic attributes. Continue with adding a public attribute “name” to the class Person. The multiplicity of a name is 1 because for the sake of simplicity, it is assumed that every person has exactly one name. For that, 1 must be entered for the upper and lower limit. Since human names consist of letters and not numbers or other symbols, the attribute type must be a String. If the Type-field is empty (i.e., the entry is “<no value>”), you can change the type of the attribute to “<<EDataType>> <Primitive Type> String” by clicking on the button with the three dots as shown in Figure 20. In the new window expand “<<EPackage, ModelLibrary>> <Model> UML Primitive Types” and select “<<EDataType>> <Primitive Type> String”.

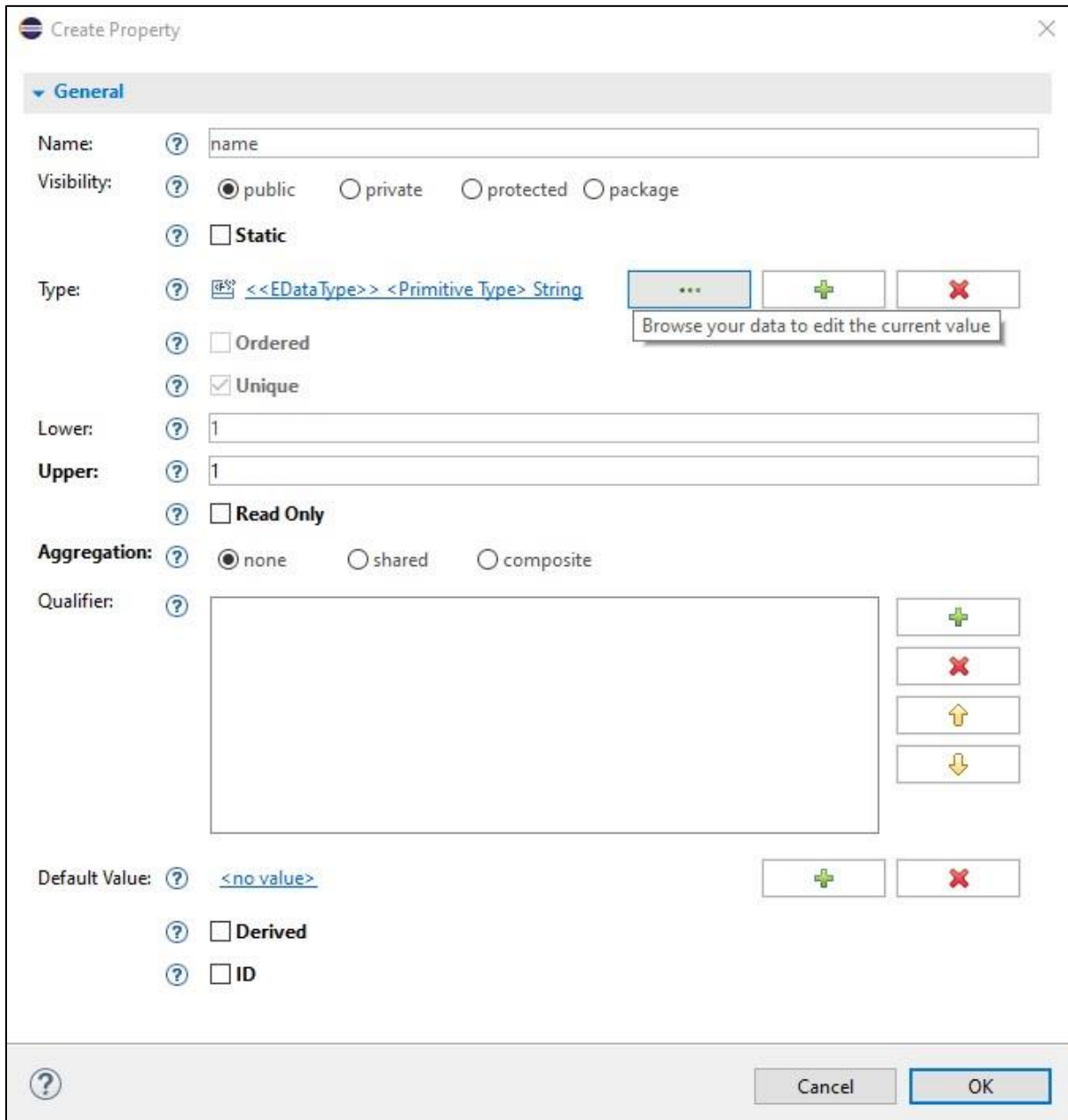


Figure 20: Creating a property

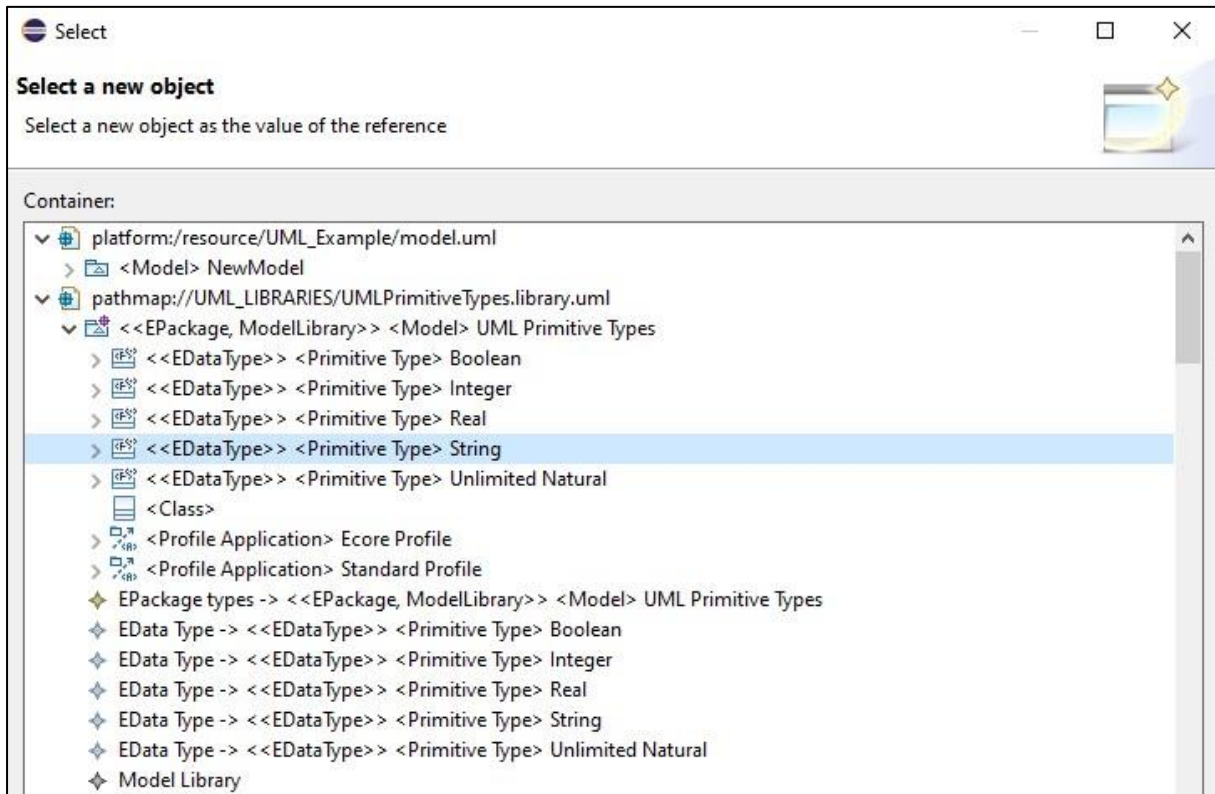


Figure 21: selecting the type of the new attribute

In case the “pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml”-section is missing, click Cancel and click OK in the “Create Property”-window. This creates an attribute without type, as seen in the left image in Figure 22. Click on the attribute to activate the text editing mode. Write “String” as seen in the right image and press enter. When noticing multiple “+” in front of the attribute name, simply remove them. After this, the pathmaps should be updated and be shown the next time you add an attribute.

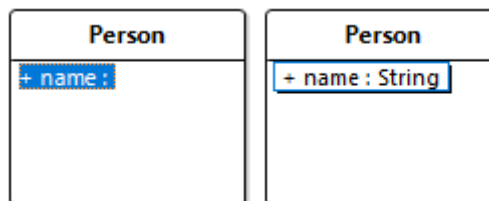


Figure 22: attribute without and with type

Three more attributes are required: A String “address” for the class “Building”, a String “purpose” for the class “PublicBuilding” and a String “cityName” for the class “City”. Remember to set upper and lower bounds to 1, when creating the attributes. Your model should then look like the one shown in Figure 23.

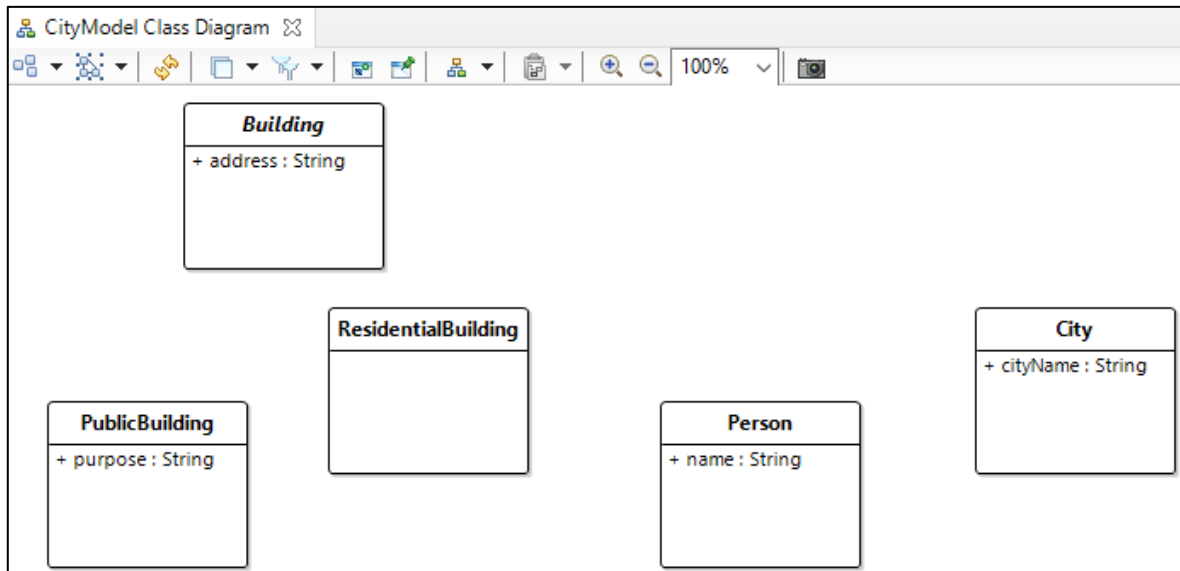


Figure 23: Model with basic attributes only

2.4.2 Adding generalizations and associations

After that, you should add generalization and association relationships, by selecting “Generalization” or “Association” in the bottom area of the Palette window. Then click on the two classes that you want to create a generalization or association between. In this example model, you must create five associations between “City” and “Person”, “City” and “PublicBuilding”, “City” and “ResidentialBuilding”, “Person” and “PublicBuilding”, and “Person” and “ResidentialBuilding”. These associations will be edited in detail later. Then, select “Generalization” and connect “PublicBuilding” and “Building” as well as “ResidentialBuilding” and “Building”. This changes “Building” to be a superclass of both “PublicBuilding” and “ResidentialBuilding”, meaning they both have the attribute “address”. Hence it is important to first click on the “PublicBuilding” class and then on “Building”, because else “Building” would be the subclass of “PublicBuilding”. Depending on how you have arranged your classes, your model may look somewhat like the one shown in Figure 24.

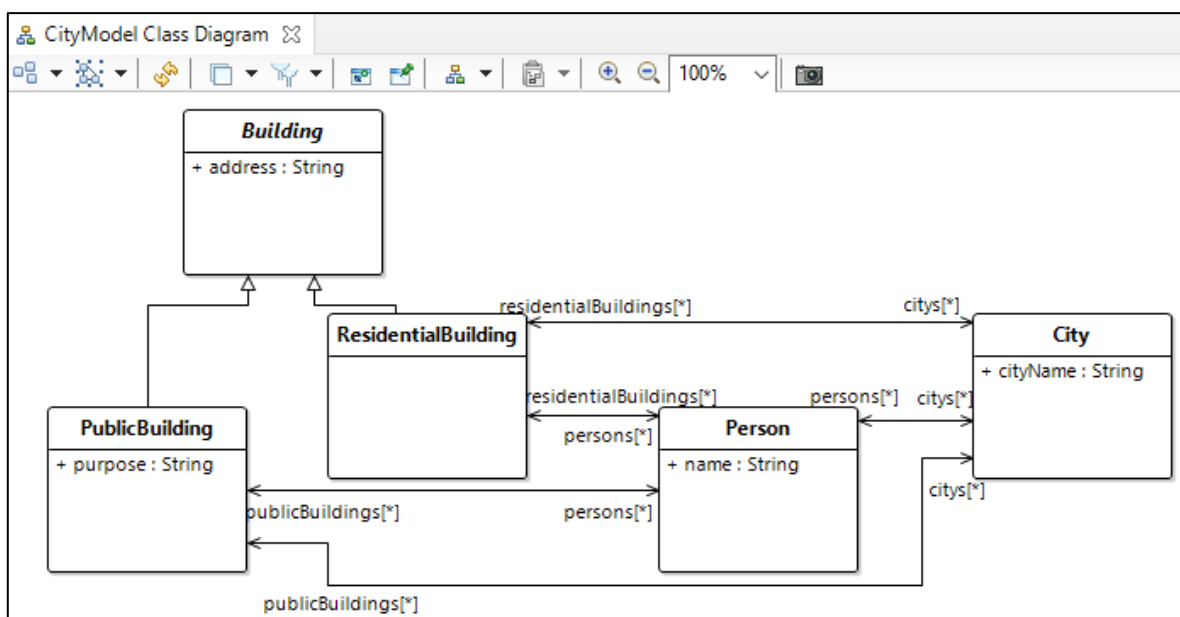


Figure 24: Model with generalizations and associations

The multiplicity of the associations must be adjusted because up to now, for example, every building can appear in several cities and every person lives in many different buildings, indicated by the small asterisk in brackets. To change the multiplicity, simply click on a connection. The "Properties" window, that can be found in the lower part of the screen, will then change its description to "<Association> ..." followed by the name of the association. Click on "General" on the left and under "Ends" the upper and lower limit of the multiplicity can be set. This is shown in Figure 25.

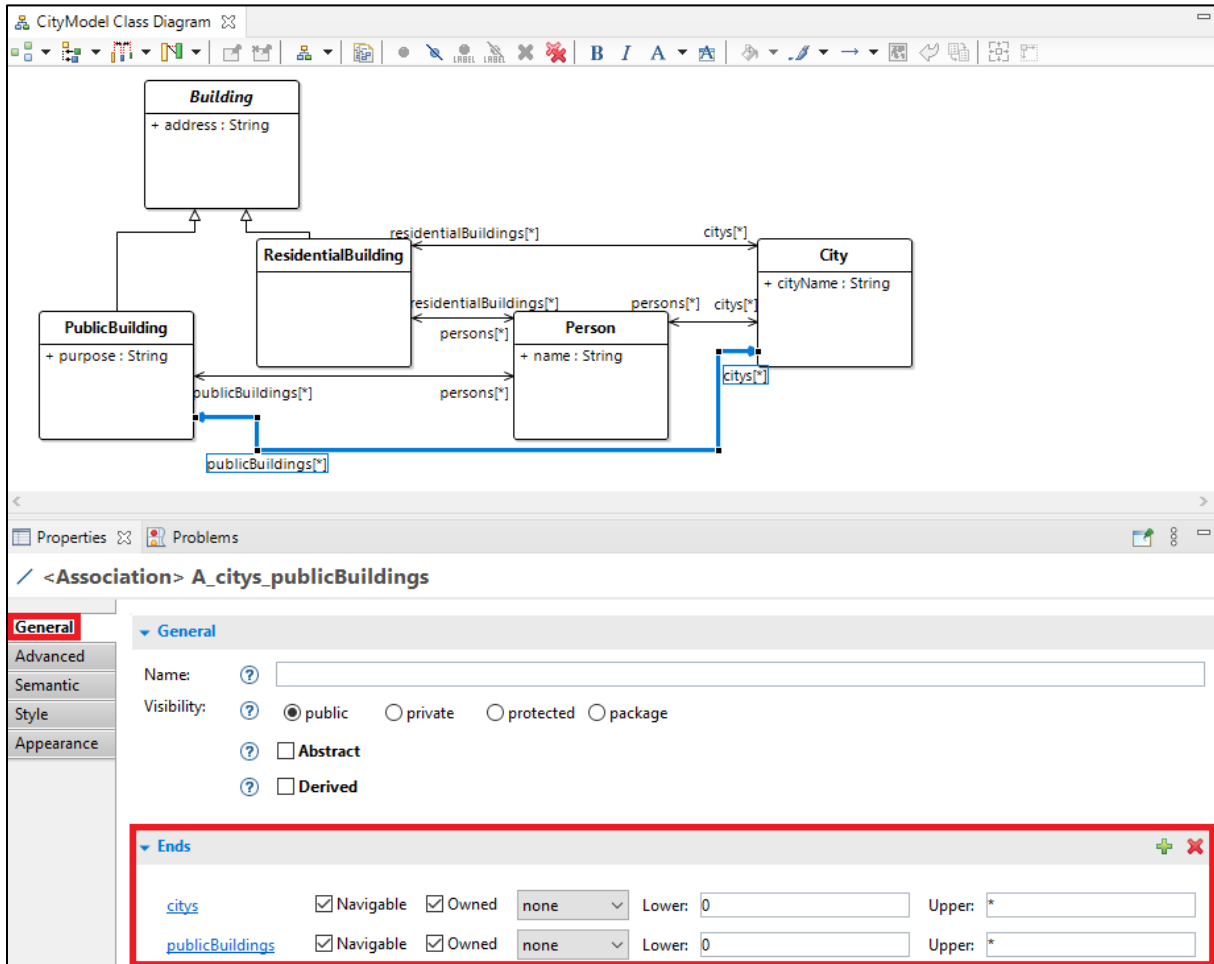


Figure 25: Changing the multiplicity

By clicking on the name of one of the ends in the "Properties"-window, a new window opens, where you can change the name of the end, as shown in Figure 26.

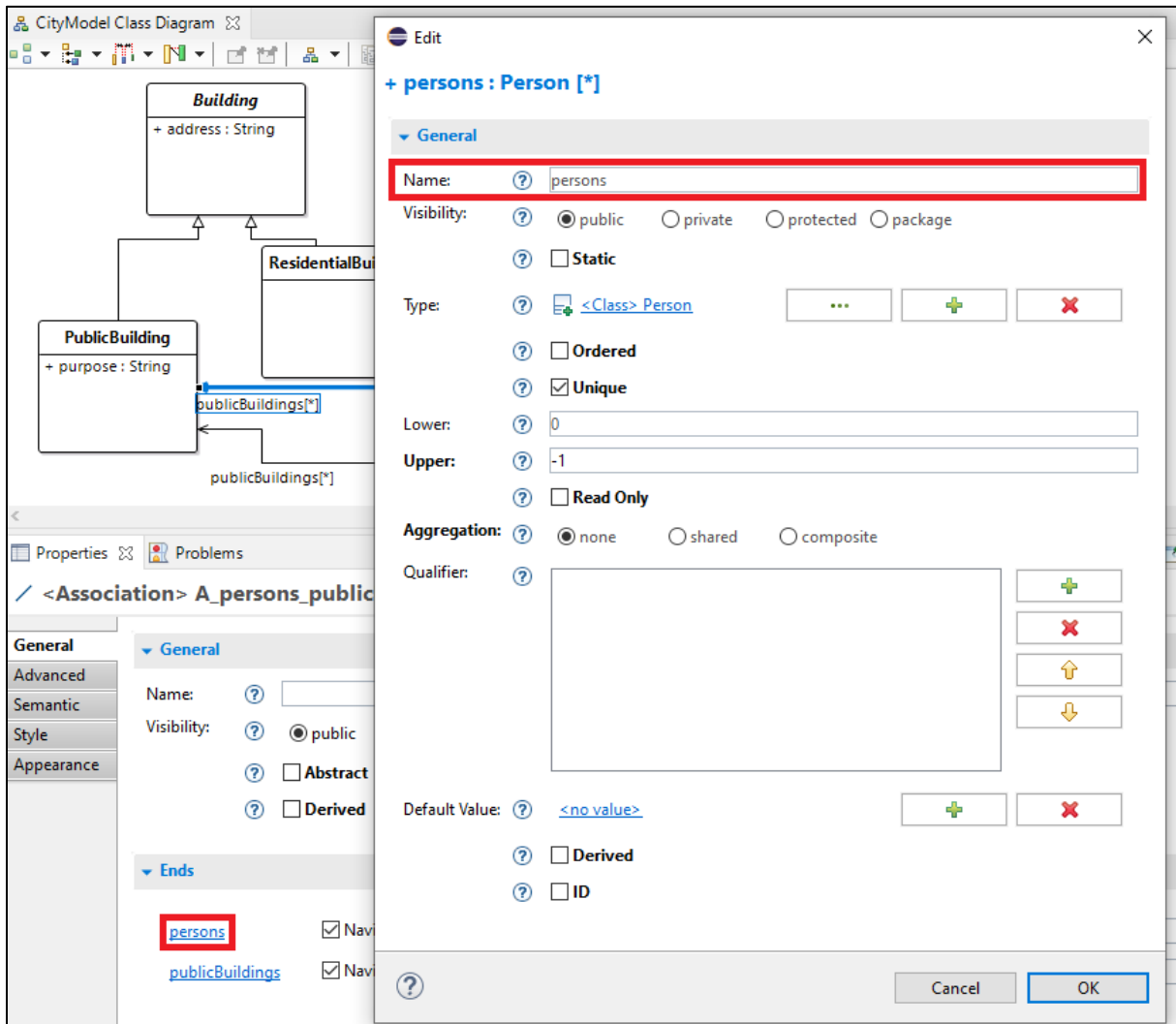


Figure 26: Changing the name of the ends of the associations

In this way change the multiplicity of the connections of the class City at the ends with the name "citys". Since every building is in exactly one city and every person lives in exactly one city, the lower and upper limit must be 1. After changing the upper and lower limit, the [*] after the end names should disappear in the diagram. Also, in the connection of Person and PublicBuilding, change the name "persons" to "employees" and the name "publicBuildings" to "worksAt" in the way described above, as well as in the connection of Person and ResidentialBuilding, change the name "persons" to "inhabitants" and "residentialBuildings" to "livesIn". Then set the lower bounds of the multiplicity of "worksAt" and "livesIn" to 0 and the upper bounds to 1 since every person can be employed in at most one public building and can live in at most one residential building. The multiplicity of "employees" and "inhabitants" can be left as it is because every public building can hold 0 to many employees and every residential building can have 0 to many inhabitants. Your model then should look like the one shown in Figure 27.

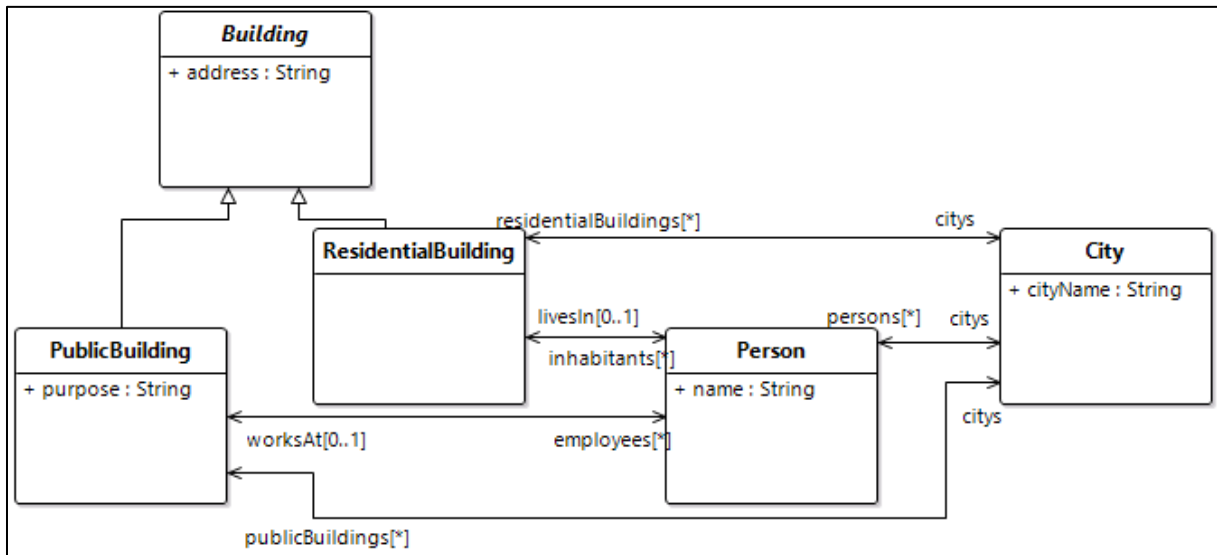


Figure 27: Adjusting the connections

You can also change the plain associations into compositions or aggregations. For that, click in the “Properties”-window in the “Ends”-section on “none” and change it either to “composite” or “shared”. In this example model, change the connections going into the class City to a composition by changing the opposing end into “composite”, as shown in Figure 28. A small black rhombus (or blue, if the connection is selected) will then appear at the end going into “City”.

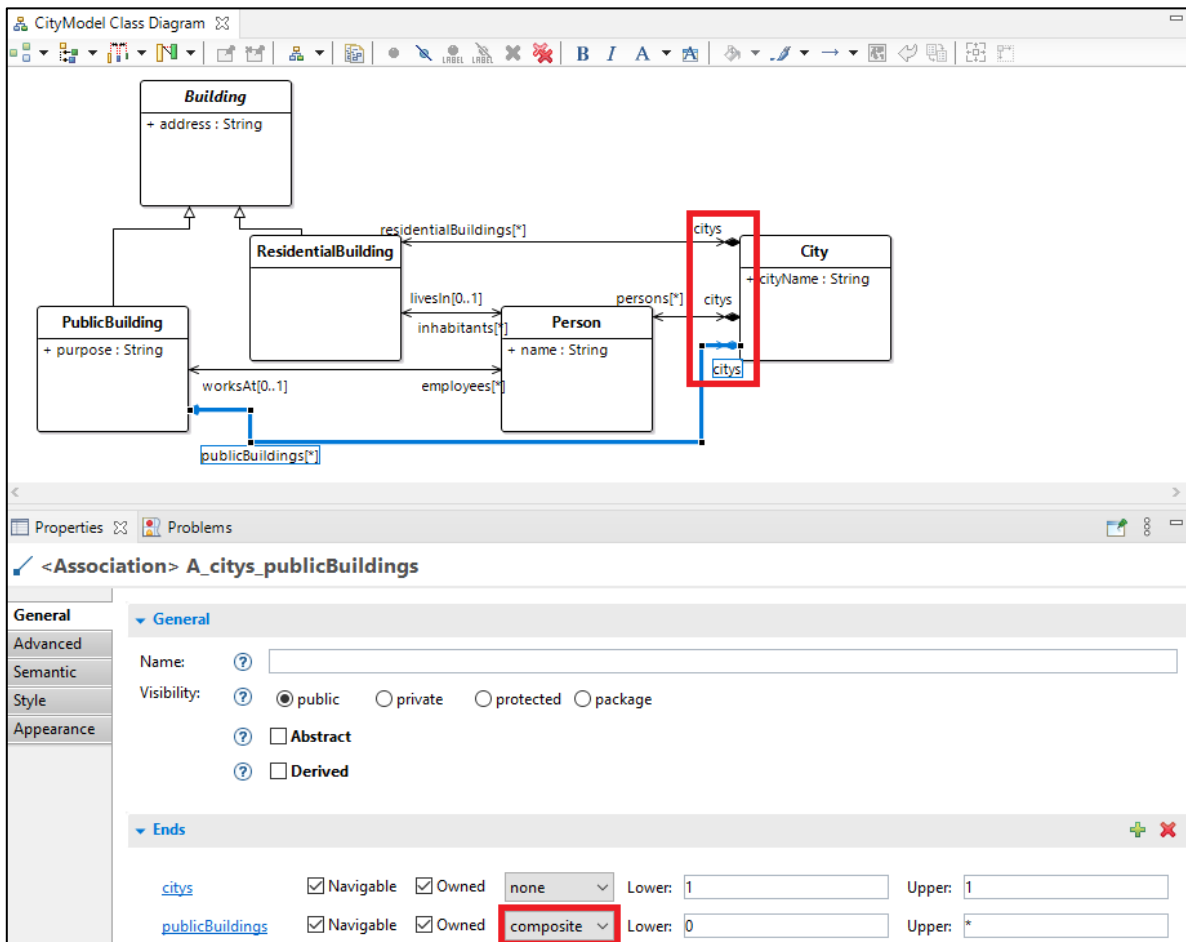


Figure 28: Changing associations into compositions

Another useful feature is the deactivation of the "Owned" fields for connections which can be done in the properties window left to the "none"/"composite"/"shared" options. This will add an attribute with the name of the end in the adjacent class. At the same time, the arrows at the opposite ends can be deactivated by unchecking "Navigable" to create a unidirectional association. In this example this is done for all associations and compositions. Remove the arrows at the connections going into "City" by deselecting the "Navigable" option for "citys" in the three compositions as shown in Figure 29 as well as deselecting the "Owned" option for the opposing end. This adds the three attributes "residentialBuildings", "publicBuildings" and "persons" to "City". Repeat for the connections between "PublicBuilding" and "Person" as well as "ResidentialBuilding" and "Person" so that the attribute "employees" is added to "PublicBuilding", and the attribute "inhabitants" is added to "ResidentialBuilding" and the arrows at the opposing ends are removed. The result is shown in Figure 29.

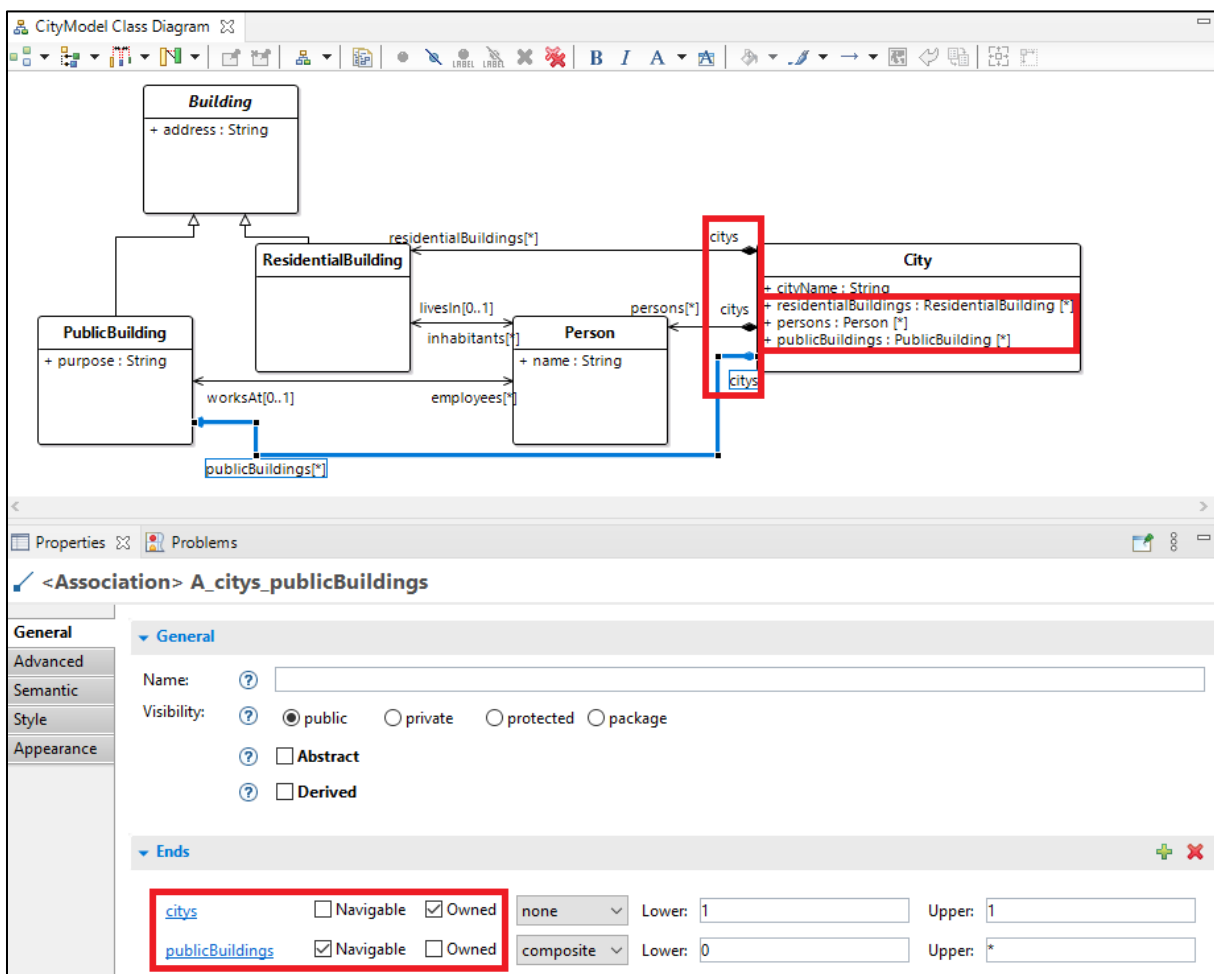


Figure 29: Changing the "Navigable" and "Owned" options

2.4.3 Adding functions

The classes and their attributes are connected to each other by the associations, but so far (apart from the getter and setter methods) there is no use for them. Hence, some methods should be added in the next step. To do this, click on "Operation" in the palette on the right and then on a class. A new window opens, where you can set the name of the operation and make the method static or abstract as well as creating a query method (see Figure 30). Then click OK to add the method to the chosen class.

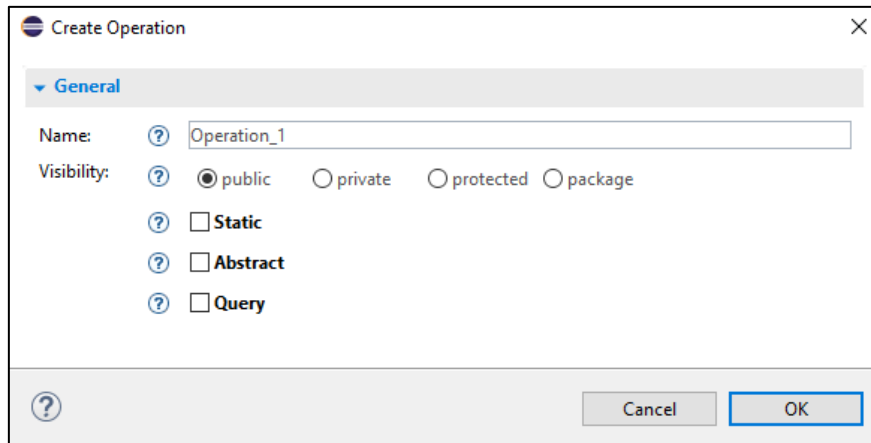


Figure 30: Creating an operation

In this way add three methods named "printCityContents", "changeResidence" and "changeWorkplace" to the class City. Continue by adding two methods "addResident" and "removeResident" to "ResidentialBuilding" as well as two methods "addEmployee" and "removeEmployee" to "PublicBuilding". You can add parameters to the created methods by clicking on the green plus symbol to the right of the "Parameters" section in the "General" tab in the "Properties" window when a method is selected. A new window will open asking for the type of the new object. Select "<Parameter>" (if it is not already selected) and click "Finish" as shown in Figure 31. This will create an unnamed parameter without a type.

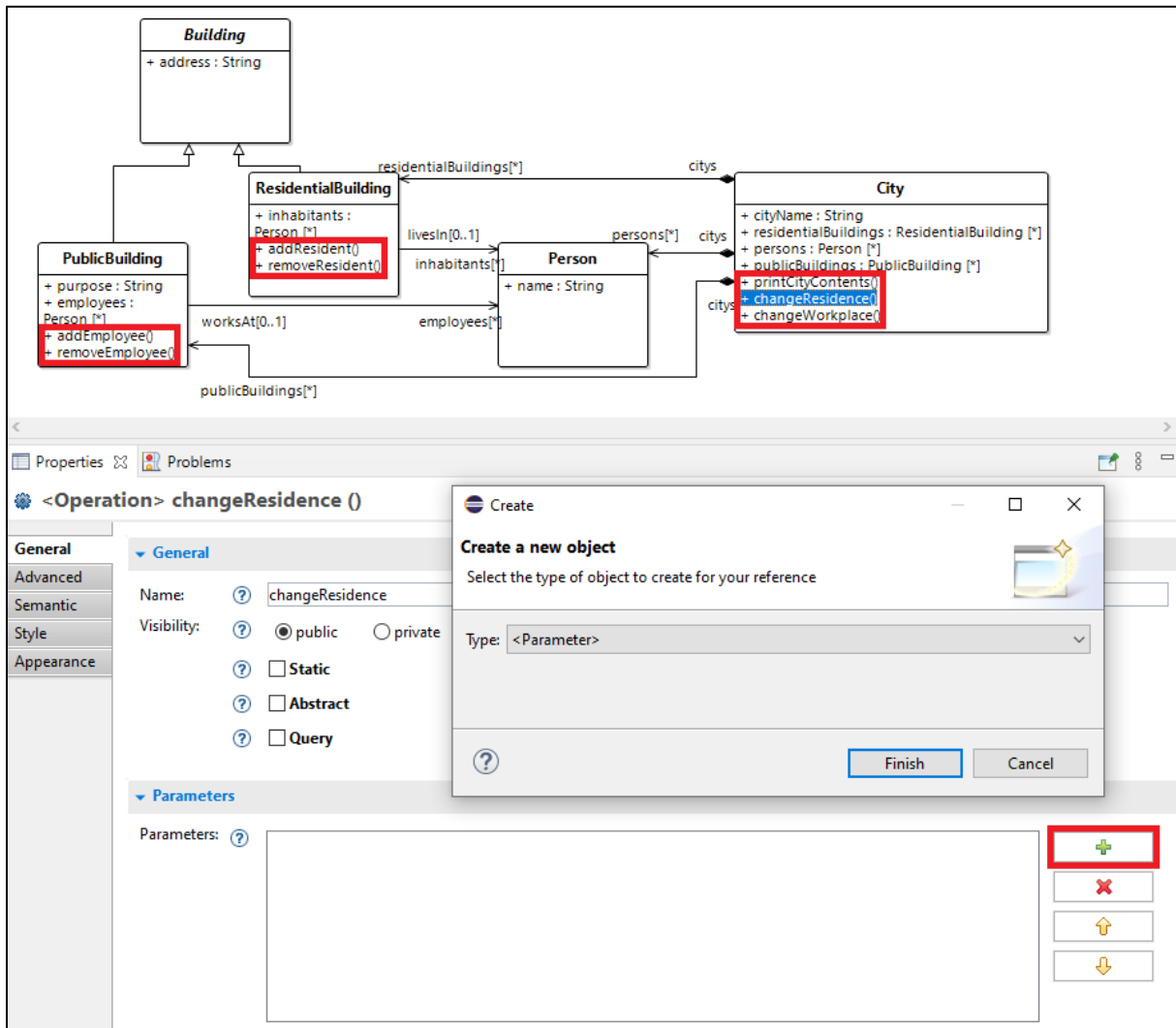


Figure 31: Adding parameters to the newly created methods

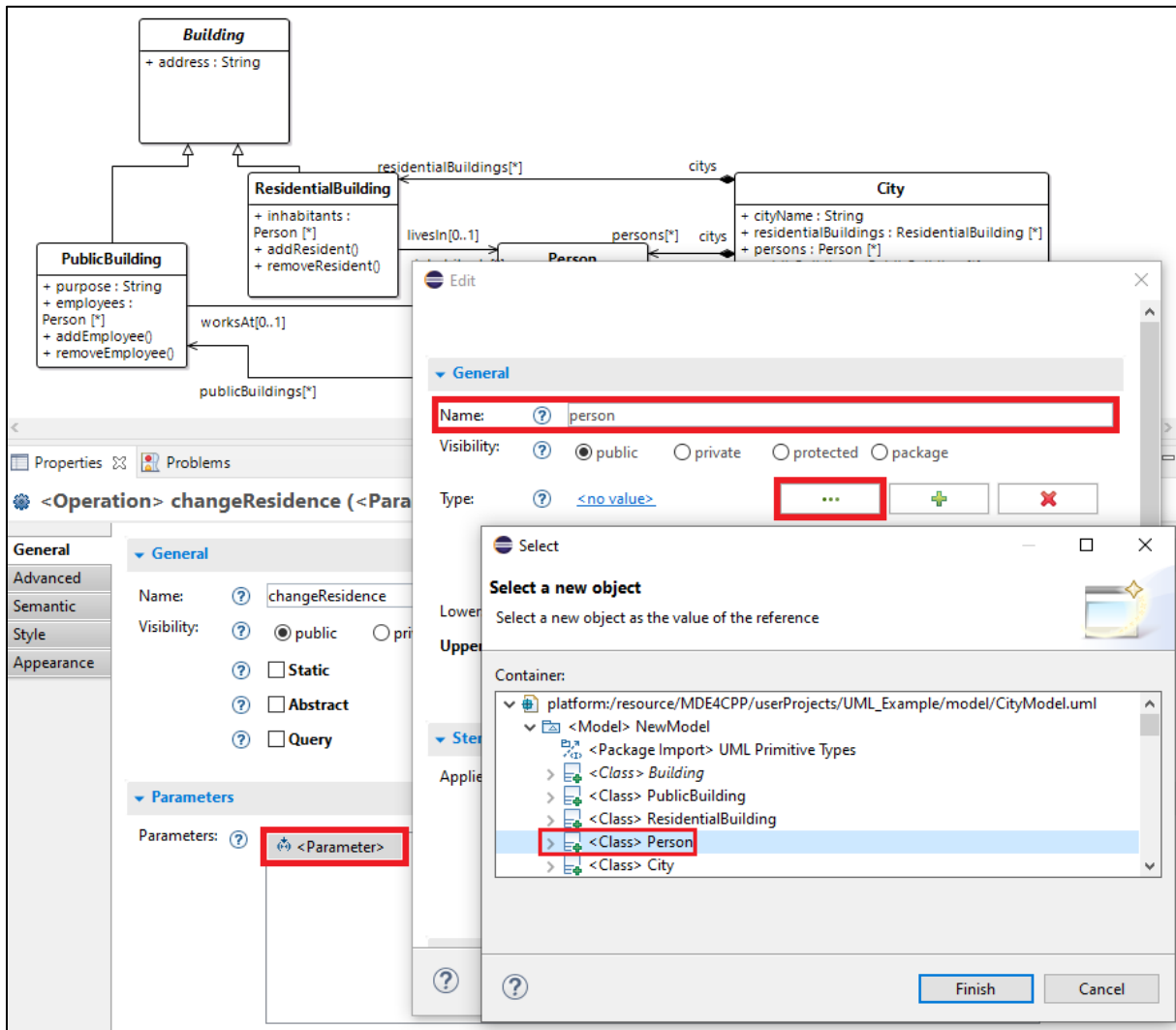


Figure 32: Customizing a parameter

To change the name and type of a selected parameter, simply double-click on the parameter in the “General” tab in the “Properties” window. A new window will open, where you can change the name in the top box and the type by clicking on the three dots to the right of the “Type”-field, as shown in Figure 32. When clicking on the dots, a new window will open, where you can select a type. In this example model, only classes of the model and UML primitive types are used. These can be selected, when expanding “<Model> NewModel” or “<<EPackage, ModelLibrary>> <Model> UML Primitive Types” and clicking on the corresponding names, as shown in Figure 32. Continue by adding a parameter “person” with the type “Person” as described to the methods “addEmployee”, “removeEmployee”, “addResident”, “removeResident”, “changeResidence” and “changeWorkplace”, as well as two String parameters “residentialBuildingAddress” to “changeResidence” and “publicBuildingAddress” to “changeWorkplace”. The result can be seen in Figure 33. As you can see, the parameters in brackets have the specified name and type.

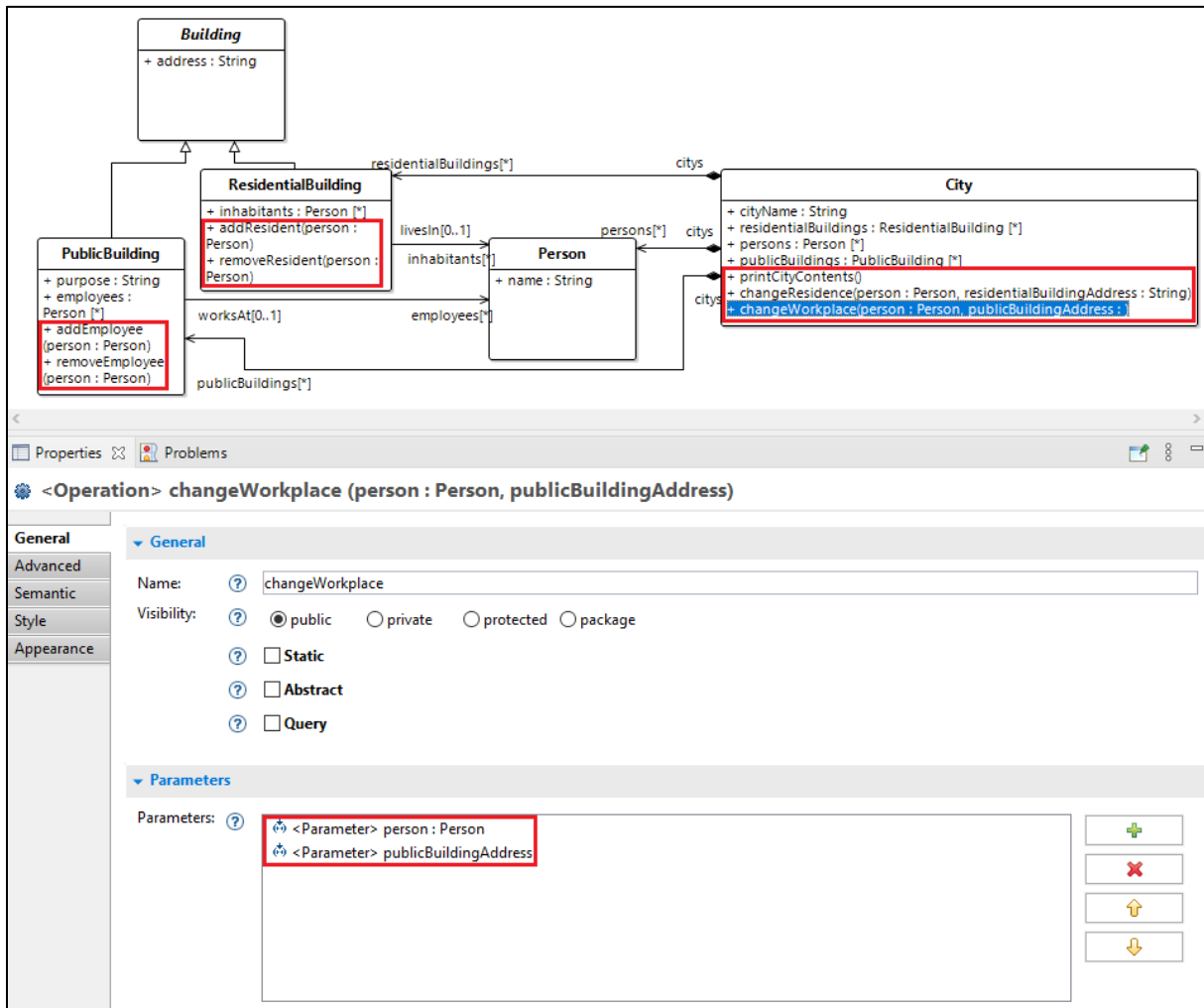


Figure 33: Methods with customized parameters

2.4.4 Implementing function behavior

The created methods now have a name and parameters, but nothing happens if they are called, because no function behavior is specified yet. For that, go to the “Advanced” tab in the “Properties” window and again, click on the green plus symbol on the right of the “Method” section. A new window will open asking for a container to create a new behavior. Expand the “<Model> NewModel” section and click on the class where you want to add the function behavior. Also, change the Type at the bottom from “<Activity>” to “<Function Behavior>”. Repeat this step for each of the seven created methods of the example model. The container class for the function behavior is the function’s class, like it is shown in Figure 34. Do not forget to change the Type to “<Function Behavior>”. Then click finish. The function behavior can then be seen in the “Method” section of the “Advanced” tab in the “Properties” window. Double-clicking on it opens a new window, where you can give an appropriate name. Save the model afterwards.

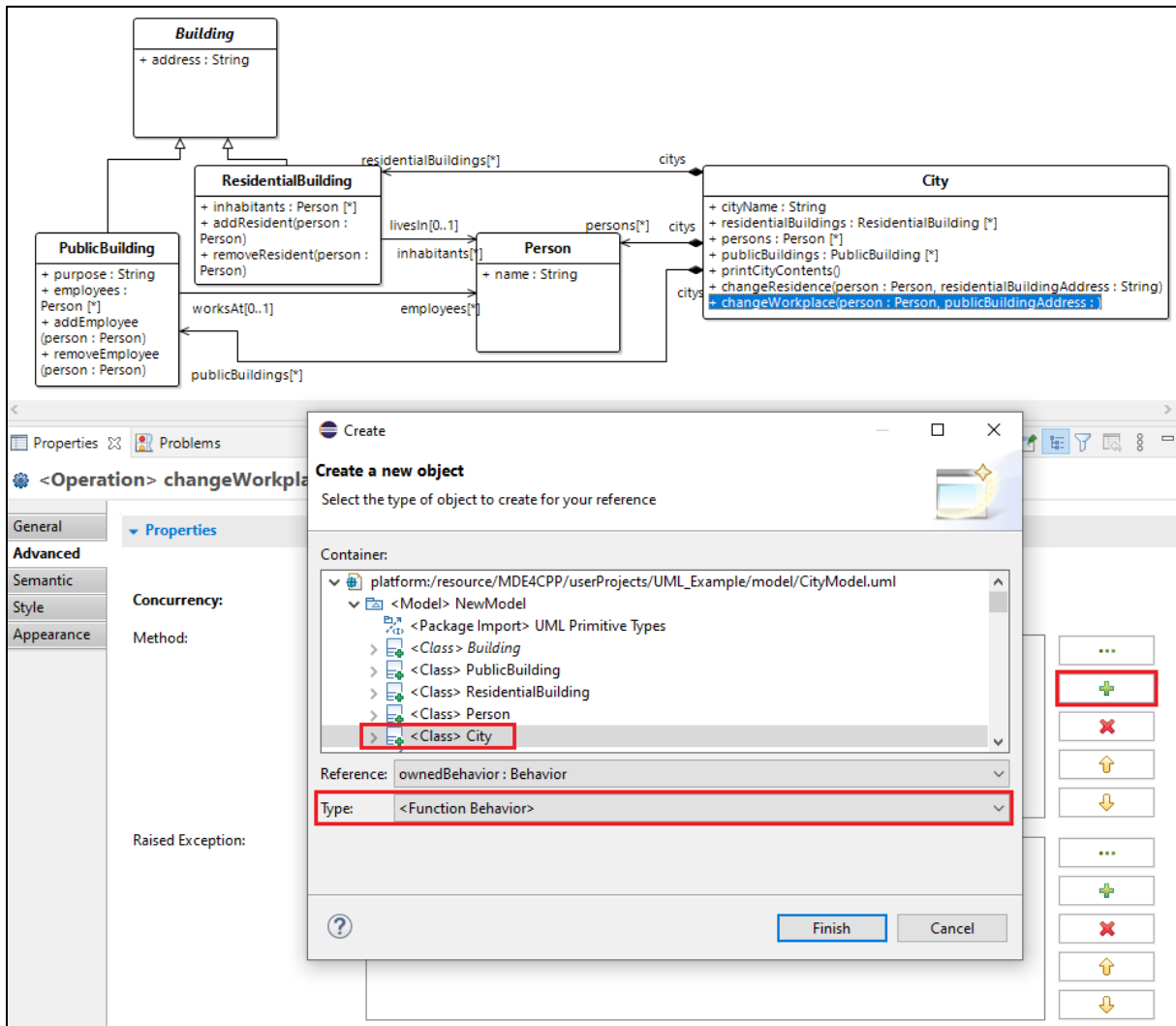


Figure 34: Adding function behavior

After this step, expand “UML_Example” in the menu in the Model Explorer on the left. Double-clicking on CityModel.uml opens the UML file containing all classes, attributes, functions, and associations. Expand the first line, which should look like “platform:/resource/MDE4CPP/userProjects/UML_Example/CityModel.uml” and expand the class containing the function behavior. In the example model, the class City holds the function behavior “changeWorkplace”, “changeResidence” and “printCityContents”. Selecting a function behavior displays some options in the “Properties” window. The two important lines are “Body” and “Language” as shown in Figure 35 below.

2 Creating an UML Diagram and deriving Code using MDE4CPP

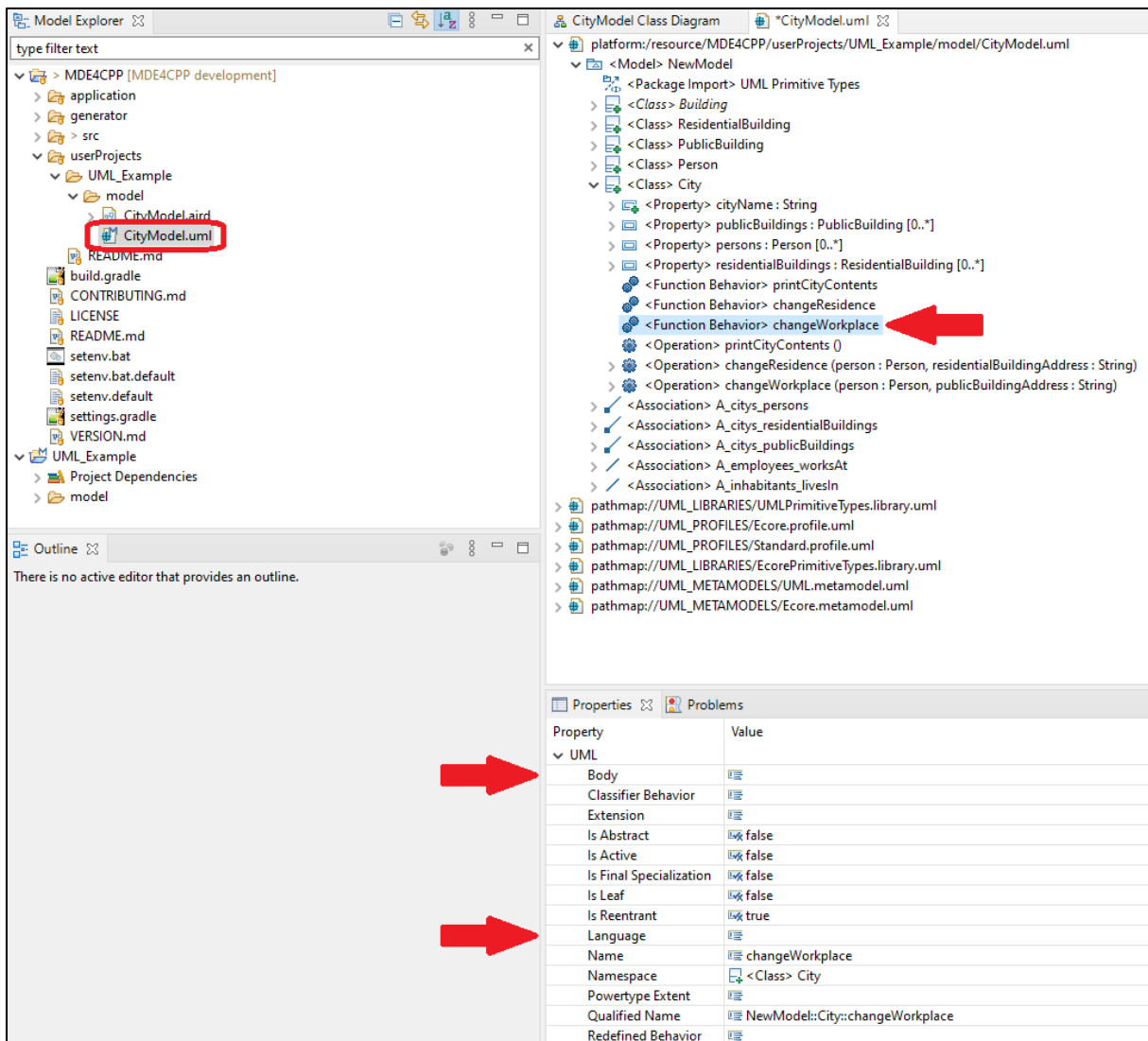


Figure 35: Setting function body and language

The next step is to fill the “Body” and “Language” lines. “Body” should contain the method implementation and the needed header files, while “Language” is used to specify which parts of “Body” contain the method implementation and which parts describe the header files. To add something to these lines, click on the three dots on the right of the “Body” line, then a new window opens, where you can enter the function code or the language value. In this example, select “changeWorkplace” in the class City. Start with the “Body” line and copy the code displayed in Code part 1 in the “Value” field:

```

//this method changes the workplace of a person

//get all public buildings and persons of the city
std::shared_ptr<Bag<PublicBuilding>> pBuildings = this -> getPublicBuildings();
std::shared_ptr<Bag<Person>> persons = this -> getPersons();

//first, we need to find the correct public building for the given address
//therefore, we iterate through all public buildings of the city
for(Bag<PublicBuilding>::const_iterator it=pBuildings->begin();it!=pBuildings->end();it++)
{
    //removing a person from a public building (his previous workplace)
    //check whether there is an employee with the same name as the input
    //(by iterating through all employees and comparing their names)
    std::shared_ptr<Bag<Person>> employees = (*it) -> getEmployees();
    for(Bag<Person>::const_iterator pers_it=employees -> begin();
        pers_it!=employees -> end(); pers_it++)
    {
        //if there is a person with the same name, he is removed from the public building
        //(his previous workplace)
        if ((*pers_it) -> getName() == person -> getName()) {
            //removing the person from the list of employees of the building
            (*it) -> removeEmployee(person);
            std::cout << "Removed employee " << person -> getName() << " from " << (*it) ->
            getAddress() << std::endl;
            break;
        }
    }
    //adding a person as employee to a building (his new workplace)
    //only if a public building with the same address as the given address exists
    if ((*it) -> getAddress() == publicBuildingAddress) {
        //adds the person to the list of employees of the building
        (*it) -> addEmployee(person);
        std::cout << "Added employee " << person -> getName() << " to " << (*it) ->
        getAddress() << std::endl;
    }
}
}

```

Code part 1: function implementation for changeWorkplace

Click on the “Add” button in the middle to store the copied code in the body and click OK. The provided code iterates through all existing public buildings of the city iterating through the list of employees of every public building to check whether the provided person is employed anywhere. If the current public building has the same address as the input address string, the person is added to the list of employees. Next, add the needed header files. Open the body window again by clicking on the three dots and write in the “Value” field:

```

#include "CityModel\PublicBuilding.hpp"
#include "CityModel\Person.hpp"

```

This will include two generated header files into the “City” class, because they are needed for the function. Click on “Add” again. The “#include ...”-lines should then be added to the “Feature” field, but below the code inserted before. Change that by clicking on the “Up” button in the middle. The “#include ...”-part should then be moved up one line, while the previously inserted code is moved down one line, like it is shown in Figure 36. Click on OK and save the changes.

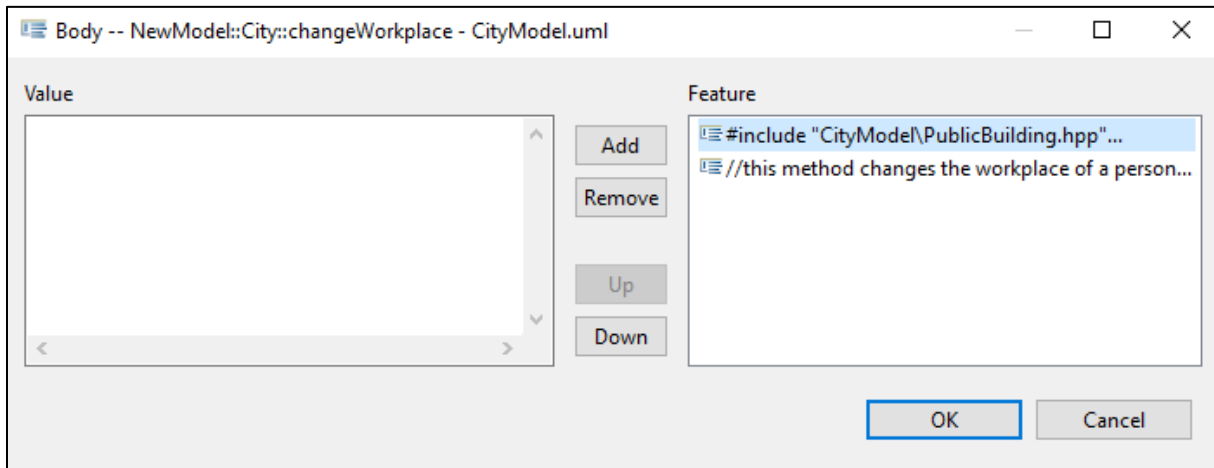


Figure 36: Adding includes and body to a function behavior

So far nothing would happen because the generator cannot do anything with the code and the includes, because the language is not specified. Go to the “Language” line in the “Properties” window and click on the three dots on the right so that the “Language” window opens. The first part of the “Body” section is formed by the includes, so you must enter “INCLUDE” (in capital letters) into the “Value” field and click on “Add”. The second part of the “Body” section contains the function code being written in C++, so you must enter “CPP” (again in capital letters) and click on “Add”. The result should look somewhat like Figure 37. Click on “OK” and save the changes. This step can be done for all created function behaviors because all functions will need includes and will have a C++ function body, irrespective of the actual content.

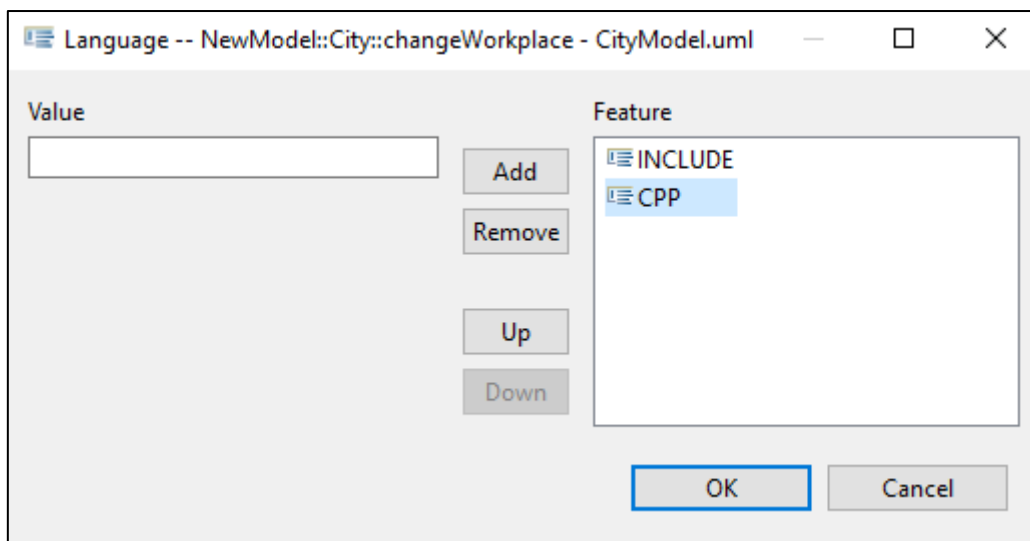


Figure 37: Setting the language

In the way described above add the following header files as includes and the code snippet shown in Code part 2 to the body of the “changeResidence” function behavior:

```
#include "CityModel\ResidentialBuilding.hpp"
#include "CityModel\Person.hpp"
```

```

//this method changes the home of a person

//all residential buildings and persons of the city
std::shared_ptr<Bag<ResidentialBuilding>> rBuildings = this -> getResidentialBuildings();
std::shared_ptr<Bag<Person>> persons = this -> getPersons();

//first, we need to find the correct residential building for the given address
//therefore, we iterate through all residential buildings of the city
for(Bag<ResidentialBuilding>::const_iterator it=rBuildings -> begin();
    it!=rBuildings -> end(); it++)
{
    //removing a person from a residential building (his previous home)
    //check whether there is a resident with the same name as the input
    //(by iterating through all residents and comparing their names)
    std::shared_ptr< Bag < Person > > residents = (*it) -> getInhabitants();
    for(Bag<Person>::const_iterator pers_it=residents -> begin();
        pers_it!=residents -> end(); pers_it++)
    {
        //if there is a person with the same name, he is removed from the residential
        //building (his previous home)
        if ((*pers_it) -> getName() == person -> getName()) {
            //removing the person from the list of residents of the building
            (*it) -> removeResident(person);
            std::cout << "Removed resident " << person -> getName() << " from " <<
                (*it) -> getAddress() << std::endl;
            break;
        }
    }
    //adding a person as resident to a building (his new home)
    //only if a residential building with the same address as the given address exists
    if ((*it) -> getAddress() == residentialBuildingAddress) {
        //adds the person to the list of residents of the building
        (*it) -> addResident(person);
        std::cout << "Added resident " << person -> getName() << " to " <<
            (*it) -> getAddress() << std::endl;
    }
}
}

```

Code part 2: function implementation for changeResidence

The provided code iterates through all existing residential buildings of the city iterating through the list of inhabitants of every residential building to check whether the provided person is residing anywhere. If the current residential building has the same address as the input address string, the person is added to the list of inhabitants of this residential building.

In the same way, add the following header files to the “printCityContents” function behavior in the “City” class, as well as the code snippet shown in Code part 3:

```

#include "CityModel\Building.hpp"
#include "CityModel\Person.hpp"

```

The provided code in Code part 3 iterates through all public buildings, residential buildings and persons existing in the city and prints their attributes to the console, separated by dashed lines for better readability. Remember to add “INCLUDE” and “CPP” to the “Language” section of the “Properties” window.

```

//every existing object of the city must be printed
//therefore, we need to collect all objects first
std::shared_ptr< Bag<PublicBuilding> > publicBuildings = this ->getPublicBuildings();
std::shared_ptr< Bag<ResidentialBuilding> > residentialBuildings = this ->
    getResidentialBuildings();
std::shared_ptr< Bag<Person> > persons = this -> getPersons();

std::cout << "======" << std::endl;
std::cout << "City " << this -> getCityName() << " contains: " << std::endl;
std::cout << "======" << std::endl;

std::cout << "Public Buildings:" << std::endl;
std::cout << "-----" << std::endl;

//iterating through all public buildings, printing its address, purpose, and all employees
for(Bag<PublicBuilding>::const_iterator it=publicBuildings -> begin();
    it!=publicBuildings -> end(); it++)
{
    std::cout << "\tPublic Building with address \"" << (*it) -> getAddress() <<
        "\"" << " and purpose \"" << (*it) -> getPurpose() << "\"" << std::endl;
    std::cout << "\t\tEmployees:" << std::endl;
    //list of employees in the current public building
    std::shared_ptr< Bag<Person> > persons = (*it) -> getEmployees();
    //iterating through the list of employees in the current public building
    //and printing their names
    for(Bag<Person>::const_iterator itPerson=persons -> begin();
        itPerson!=persons -> end(); itPerson++)
    {
        std::cout << "\t\t\tPerson " << (*itPerson) -> getName() << std::endl;
    }
}

std::cout << "Residential Buildings:" << std::endl;
std::cout << "-----" << std::endl;

//iterating through all residential buildings, printing its address and all residents
for(Bag<ResidentialBuilding>::const_iterator it=residentialBuildings -> begin();
    it!=residentialBuildings -> end(); it++)
{
    std::cout << "\tResidential Building with address \"" << (*it) -> getAddress() <<
        "\"" << std::endl;
    std::cout << "\t\tResidents:" << std::endl;
    //list of residents in the current residential building
    std::shared_ptr< Bag<Person> > persons = (*it) -> getInhabitants();
    //iterating through the list of residents in the current residential building
    //and printing their names
    for(Bag<Person>::const_iterator itPerson=persons -> begin();
        itPerson!=persons -> end(); itPerson++)
    {
        std::cout << "\t\t\tPerson " << (*itPerson) -> getName() << std::endl;
    }
}

std::cout << "Persons in city:" << std::endl;
std::cout << "-----" << std::endl;

//iterating through all persons of the city and printing their names
for(Bag<Person>::const_iterator it=persons -> begin(); it!=persons -> end(); it++)
{
    std::cout << "\tPerson with name: " << (*it) -> getName() << std::endl;
}
std::cout << "======" << std::endl;
std::cout << "Printed everything from City " << this -> getCityName() << std::endl;
std::cout << "======" << std::endl;

```

Code part 3: function implementation for printCityContents

Expand the “PublicBuilding” class and click on the “addEmployee” function behavior. Add the following line (that includes the header file “Person.hpp” into the function) to the “Body” section:

```
#include "CityModel/Person.hpp"
```

This header file is needed for all the remaining function behaviors (addEmployee, removeEmployee, addResident, removeResident), so you can just copy and paste it into the four remaining function behavior's bodies. After clicking on the "Add" button, copy the code snippets shown below in Code part 4, Code part 5, Code part 6, and Code part 7 to the corresponding function behaviors.

```
m_employees -> add(person);
```

Code part 4: function implementation for addEmployee

```
m_employees -> erase(person);
```

Code part 5: function implementation for removeEmployee

```
m_inhabitants -> add(person);
```

Code part 6: function implementation for addResident

```
m_inhabitants -> erase(person);
```

Code part 7: function implementation for removeResident

As shown in Figure 38 exemplary for the "addEmployee" function behavior, the header files must be specified first and then the function body. Remember to add "INCLUDE" and "CPP" to the "Language" field.

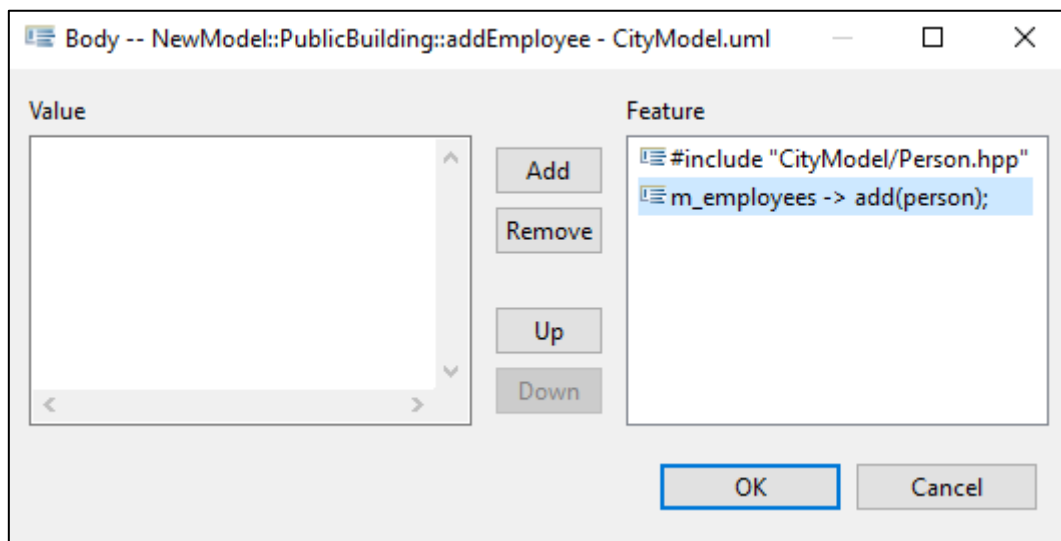


Figure 38: Function behavior body of addEmployee

2.4.5 Creating a main function

To finally execute the generated program, a main function is required. To achieve this, stereotypes with keywords are used, which are explained below. The first step is to import the UML4CPPProfile (that contains the stereotypes) into the CityModel.uml file. For that, right-click in the UML file and select "Load Resource..." which should be the third entry from the bottom in the menu. You could either click on "Browse Workspace..." and select UML4CPPProfile.uml which can be found at the path shown in Figure 39. You could also manually enter the path.

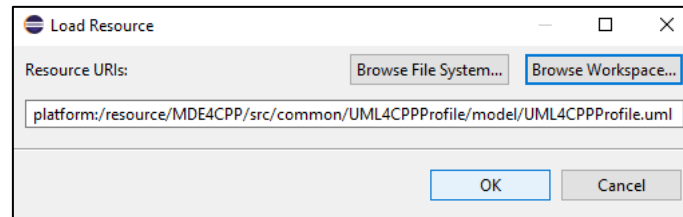


Figure 39: Importing the UML4CPPProfile

If everything went well, another item appears near the bottom of the list of items with a name similar to "platform:/resource/.../UML4CPPProfile.uml" in the UML file. Expand it by clicking on the small arrow on the left. This will show a list of stereotypes that can be applied to any UML model, as shown in Figure 40.

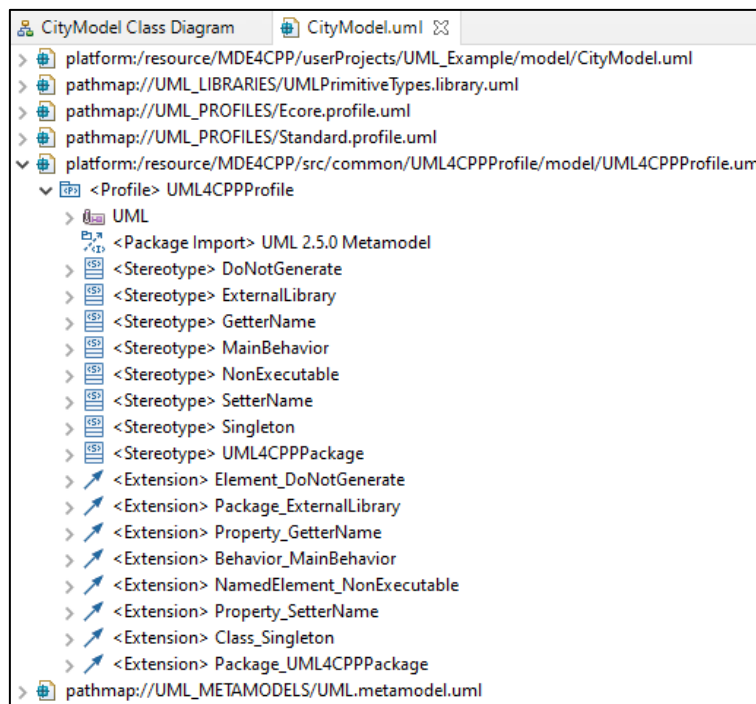


Figure 40: Stereotypes in UML4CPPProfile.uml

The required stereotype is the one named "MainBehavior". When applied to a function, it will be the main function of the generated program. The chosen behavior will then be executed by the main function. Besides this there are 5 other stereotypes:

- DoNotGenerate can be used to indicate to the generator that this element should not be generated.
- ExternalLibrary can be used to indicate that a package represents an external library. This could be applicable for example, when an external library should be used in the generated model code and is represented by an interface model while modeling, for example to be able to use external types.
- GetterName and SetterName can be used to define a special name for the getter and setter method of a property.
- NonExecutable can be used to indicate that no execution class should be generated for an element (e.g., no execution class for an opaque behavior or operation). This will make the corresponding model element non-executable for model execution.

- Singleton can be used to indicate that a class should be generated as a singleton.
- UML4CPPPackage can be used to specify additional information of a Package to parameterize the code generation process through additional properties:
 - "eclipseURI : String": the URI for referencing the original model file. Used for saving and loading persistent models during runtime. Can be an Eclipse 'PATHMAP' or an absolute path.
 - "ignoreNamespace : Boolean": indicates that the namespace of the corresponding Package should be ignored during generation.
 - "packageOnly : Boolean": indicates that only the model package should be generated (excluding classes, enumerations etc.).

To add a stereotype to the model, you must apply the profile containing the stereotype first. Hence, select the package (here it is “<Model> NewModel”), as highlighted in Figure 41. In the menu bar on the top, click on “UML Editor”, then “Package” and “(Re-)Apply Profile...”. Select “UML4CPPProfile – UML4CPPProfile.uml” as shown in Figure 42, click “Apply” and click “OK”.

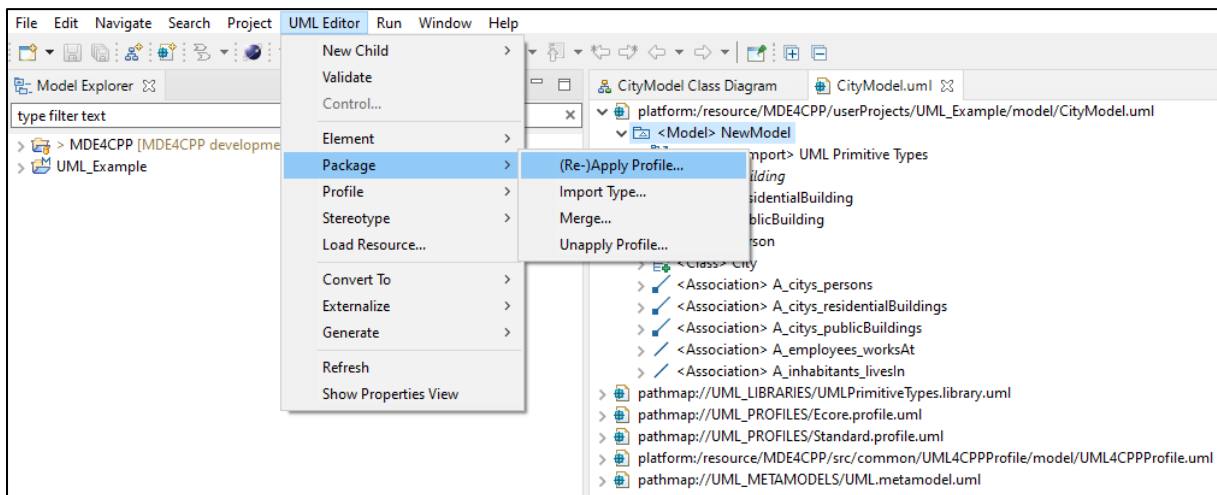


Figure 41: Applying a profile (1)

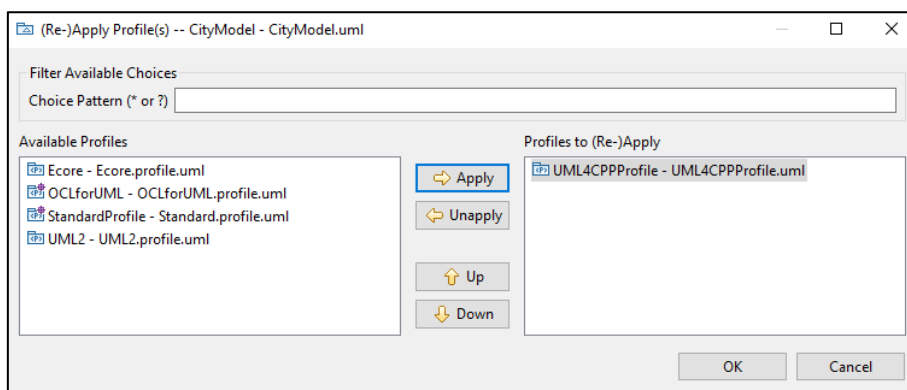


Figure 42: Applying a profile (2)

The next step is to create the function that should be marked as MainBehavior. In the CityModel.uml, right-click on “<Model> NewModel”, select “New Child”, then “Packaged Element” and select “Function Behavior”. This adds an empty “<Function Behavior>” item to the model. In the “Properties” window, you can give an appropriate

name. In the example model, the function behavior is named “mainProgram” as shown in Figure 43. Then click on “UML Editor” in the menu bar on the top. This time, select “Element” and “Apply Stereotype...” which opens a new window, as shown in Figure 44. In the left window, which is called “Applicable Stereotypes”, all stereotypes compatible with this element are listed. Select the stereotype(s) you want to add to the selected element. To make the created function the main function, select “UML4CPPProfile::MainBehavior – UML4CPPProfile.uml”, click “Apply” and then “OK”. The line of the function behavior should now look like “<<MainBehavior>> <FunctionBehavior> mainProgram”.

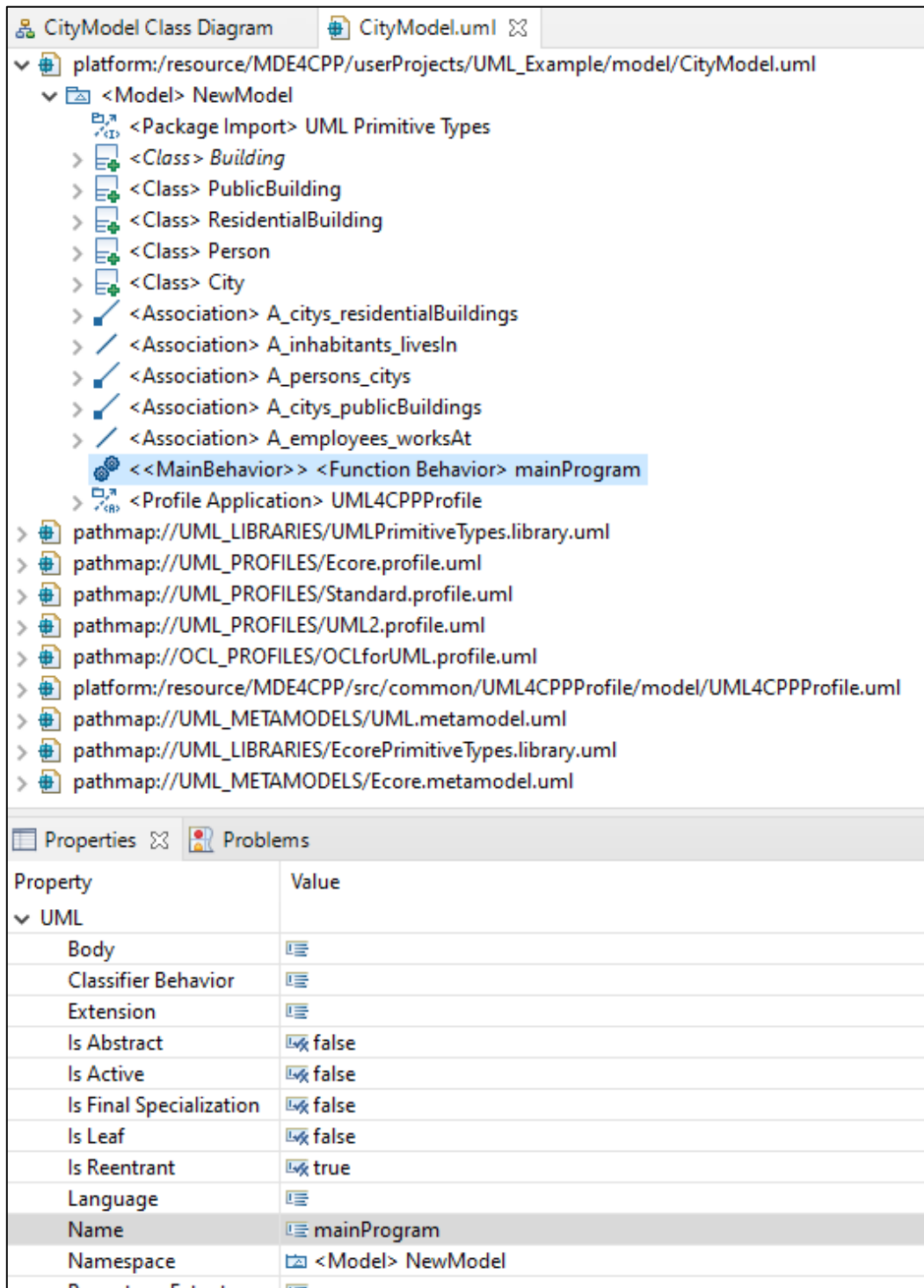


Figure 43: function behavior to be the main function

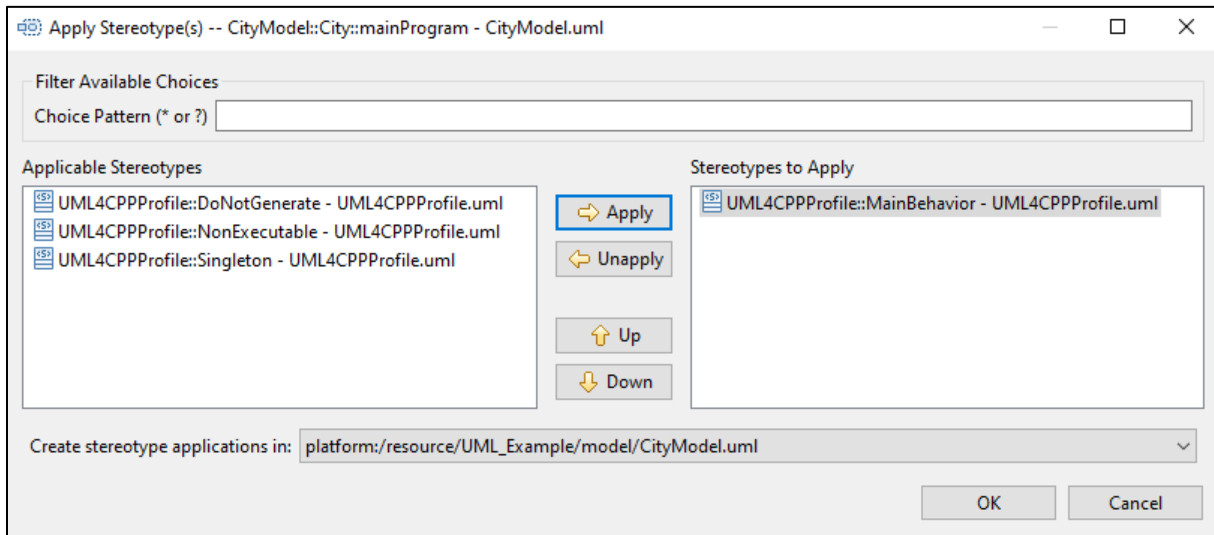


Figure 44: Applying a stereotype

Next, define the used languages in the “Language” field, which is “INCLUDE” and “CPP” as always. Then, add the needed header files to the “Body” section just like before:

```
#include "CityModel/Building.hpp"
#include "CityModel/City.hpp"
#include "CityModel/Person.hpp"
#include "CityModel/PublicBuilding.hpp"
#include "CityModel/ResidentialBuilding.hpp"
```

The goal of the main program is to accept user input and process it. The result should be for example, the creation of an object of type Person, PublicBuilding or ResidentialBuilding in the City, as well as changing attributes or to print something to the console, depending on the input. In the following, some code parts are shown that should be added to the “Value” box of the “mainProgram” function behavior’s body, starting with Code part 8. To recreate the example model, simply copy and paste the parts seamlessly. For that, start with the creation of the city:

```
//Creating a new instance of City via model factory
std::shared_ptr<City> city =factory->createCity();
```

Code part 8: main program (1)

The first user input represents the city’s name. A "cityName" variable is required for this, in which the user input is stored and then is set as name of the previously created city. The code for this can be seen in Code part 9. Copy and paste this segment right after the Code part 8.

```
//Setting city name via console input
std::cout << "Enter a city name: ";
std::string cityName;
std::getline(std::cin, cityName);
city->setCityName(cityName);
```

Code part 9: main program (2)

The next step is the main program loop, in which a user input is accepted and processed in each step until the user gives the stop command. All following code parts must be placed inside this loop at the position of the dark green, bold comment

(preferably in the order given), as can be seen in Code part 10 below. If the user types the stop command, the loop is exited, and the program terminates.

```
//Every command that is entered is stored in the input_str
//Initially, the string is empty
std::string input_str = "";

//main loop of the program, can be exited by entering "stop"
while(true) {
    //after "Enter command: " is printed, one of the following commands can be entered
    std::cout << "Enter command: ";
    std::getline(std::cin, input_str);

    //ALL FOLLOWING CODE SHOULD BE PLACED IN HERE
}
std::cout << "Program ends..." << std::endl;
```

Code part 10: main program (3)

The first command to implement is a simple “help” command printing all available commands to the console. Copy the code shown in Code part 11 into the loop at the position marked in Code part 10.

```
//when typing "help", every available command will be printed to the console
//along with a short description
if (input_str=="help") {
    std::cout << "addPerson\t\t adds a person to the city" << std::endl;
    std::cout << "addPublicBuilding\t creates a public building with an address and a purpose
in the city" << std::endl;
    std::cout << "addResidentialBuilding\t creates a residential building with an address in
the city" << std::endl;
    std::cout << "changeWorkplace\t\t changes the workplace of a person" << std::endl;
    std::cout << "changeResidence\t\t changes the place of residence of a person" <<
std::endl;
    std::cout << "print\t\t\t prints all existing objects" << std::endl;
    std::cout << "stop\t\t\t exits the program" << std::endl;
}
}
```

Code part 11: main program (4)

The first "real" command is addPerson shown Code part 12, which creates a Person object and saves it in the city. To do this, the user must enter "addPerson" and press Enter. This adds a person to the city using a factory that is created when generating the code via MDE4CPP (via the factory's method createPerson_as_persons_in_City()). In the next line, the user can then enter any character string that is assigned to the person created as a name.

```
//when typing "addPerson" and pressing enter, you can add a person to the city
//For that, you also have to type a name in the next line
else if (input_str=="addPerson") {
    //Creating a person in city via model factory
    std::shared_ptr<Person> person1 = factory->createPerson_as_persons_in_City(city);
    std::cout << "Name of person: ";
    //Setting name of the newly created person
    std::string pName;
    std::getline(std::cin, pName);
    person1->setName(pName);
}
}
```

Code part 12: main program (5)

The next two commands add a PublicBuilding or a ResidentialBuilding to the city. Therefore, the structures of Code part 13 and Code part 14 are very similar. The only major difference is that a purpose must be specified when creating a PublicBuilding. The rest works in the same way as addPerson: first the object is created as part of

the city, then the second input string is set as the address of the building. When a PublicBuilding is created, a third input string is set as the purpose.

```
//when typing "addPublicBuilding" and pressing enter, you can add a public building
//to the city
//For that, you have to type an address in the next line
//Also, you have to type a purpose in the next line (e.g. "Mall" or "Hospital")
else if (input_str=="addPublicBuilding") {
    //Creating a public building in city via model factory
    std::shared_ptr<PublicBuilding> pBuilding = factory ->
        createPublicBuilding_as_publicBuildings_in_City(city);
    std::cout << "Address of public building: ";
    //Setting address of the newly created public building
    std::string address;
    std::getline(std::cin, address);
    pBuilding->setAddress(address);
    std::cout << "Purpose of public building: ";
    //Setting purpose of the newly created public building
    std::string purpose;
    std::getline(std::cin, purpose);
    pBuilding->setPurpose(purpose);
}
```

Code part 13: main program (6)

```
//when typing "addResidentialBuilding" and pressing enter, you can add
//a residential building to the city
//For that, you have to type an address in the next line
else if (input_str=="addResidentialBuilding") {
    //Creating a residential building in city via model factory
    std::shared_ptr<ResidentialBuilding> rBuilding = factory ->
        createResidentialBuilding_as_residentialBuildings_in_City(city);
    std::cout << "Address of residential building: ";
    //Setting address of the newly created residential building
    std::string address;
    std::getline(std::cin, address);
    rBuilding->setAddress(address);
}
```

Code part 14: main program (7)

The next command “changeResidence” (shown in

Code part 15) is used to add an existing person as resident to a residential building removing it from his former home. Accordingly, it takes the address of an existing residential building as well as a person as input parameters.

```
//when typing "changeResidence" and pressing enter, you can add a person as resident
//to a residential building (and removing him from the previous residential building)
//For that, you have to type the name of the person in question in the next line
//Also, you have to type an address of a residential building in the next line
else if (input_str=="changeResidence") {
    //First input: name of person
    std::cout << "Name of resident: ";
    std::string name;
    std::getline(std::cin, name);

    //"resident = nullptr": placeholder for the pointer of the (already created) person,
    //which is to find
    std::shared_ptr<Person> resident = nullptr;
    //Get all persons from city and search for person with the same name as the input string
    std::shared_ptr<Bag<Person>> persons = city->getPersons();
    for(Bag<Person>::const_iterator it=persons->begin();it!=persons->end();it++)
    {
        std::string itName = (*it)->getName();
        //check for every person if the name is equal to the input string
        //if true: nullptr is replaced with the pointer of the found person
        if (itName == name) {
            resident = (*it);
        }
    }

    //Second input: address of building, where the person will be added
    std::cout << "Address of residential building that will be the new residence: ";
    std::string address;
    std::getline(std::cin, address);

    //true if resident has been found in the list of persons of the city
    //if true, person is added to the list of residents of the building at the address
    //given in the second input
    //if false (i.e. no person with the given name exists), a short message is printed,
    //but nothing changes besides that
    if (resident != nullptr) {
        city->changeResidence(resident, address);
    } else {
        std::cout << "Person \"" << name << "\" not found. To assign a person as resident" <<
            "to a residential building, the person must have been created before!"<< std::endl;
    }
}
```

Code part 15: main program (8)

Like the “changeResidence” command that, the next command “changeWorkplace” shown in Code part 16 is used to add an existing person as employee to a public building removing it from his former workplace. Accordingly, it takes the address of an existing public building as well as a person as input parameters.

```

//when typing "changeWorkplace" and pressing enter, you can add a person as employee
//to a public building (and removing him from the previous public building)
//For that, you have to type the name of the person in question in the next line
//Also, you have to type an address of a public building in the next line
else if (input_str=="changeWorkplace") {
    //First input: name of person
    std::cout << "Name of employee: ";
    std::string name;
    std::getline(std::cin, name);

    //"employee = nullptr": placeholder for the pointer of the (already created) person,
    //which is to find
    std::shared_ptr<Person> employee =nullptr;
    //Get all persons from city and search for person with the same name as the input string
    std::shared_ptr<Bag<Person>> persons = city->getPersons();
    for (Bag<Person>::const_iterator it=persons->begin();it!=persons->end();it++)
    {
        std::string itName = (*it)->getName();
        //check for every person if the name is equal to the input string
        //if true: nullptr is replaced with the pointer of the found person
        if (itName == name) {
            employee = (*it);
        }
    }

    //Second input: address of building, where the person will be added
    std::cout << "Address of public building that will be the new workplace: ";
    std::string address;
    std::getline(std::cin, address);

    //true if employee has been found in the list of persons of the city
    //if true, person is added to the list of employees of the building at the address
    //given in the second input
    //if false (i.e. no person with the given name exists), a short message is printed,
    //but nothing changes besides that
    if (employee != nullptr) {
        city->changeWorkplace(employee, address);
    } else {
        std::cout << "Person \"\" << name << "\"" not found. To assign a person as employee to"<<
            "a public building, the person must have been created beforehand!" << std::endl;
    }
}
}

```

Code part 16: main program (9)

The last two commands are “print” and “stop”. When “print” is called, all existing objects in the city are printed to the console in a readable format. “stop” leaves the main program loop and the program terminates. Also, if the input does not match any of the possible commands, a short message is printed, indicating “help” as a useful command. This can be seen in Code part 17, which is the last part of code for the main program. Then click “Add” and “OK” in the “Body” window.


```

//when typing "print" and pressing enter, all existing objects will be printed
//for more information, read the comments at CityImpl::printCityContents()
else if (input_str=="print") {
    city->printCityContents();
}

//when typing "stop" and pressing enter, the program's main loop will be exited
//and the program terminates
else if (input_str=="stop") {
    break;
}

//when typing anything else, a short message appears, indicating help by typing "help"
else {
    std::cout << "Unknown command! Type \"help\" for a list of supported commands.<<
    std::endl;
}

```

Code part 17: main program (10)

2.5 Generating and compiling the model

Before generating the model, make sure that the model's name is the same as the UML file's name. Like it is shown in Figure 45, the file's name is "CityModel.uml" while the model's name is "NewModel", so they do not match. Change that by clicking on "NewModel" and type "CityModel" as its new name. You could also choose every other name, but the UML file's name must match the model's name. Remember to save the changes afterwards.

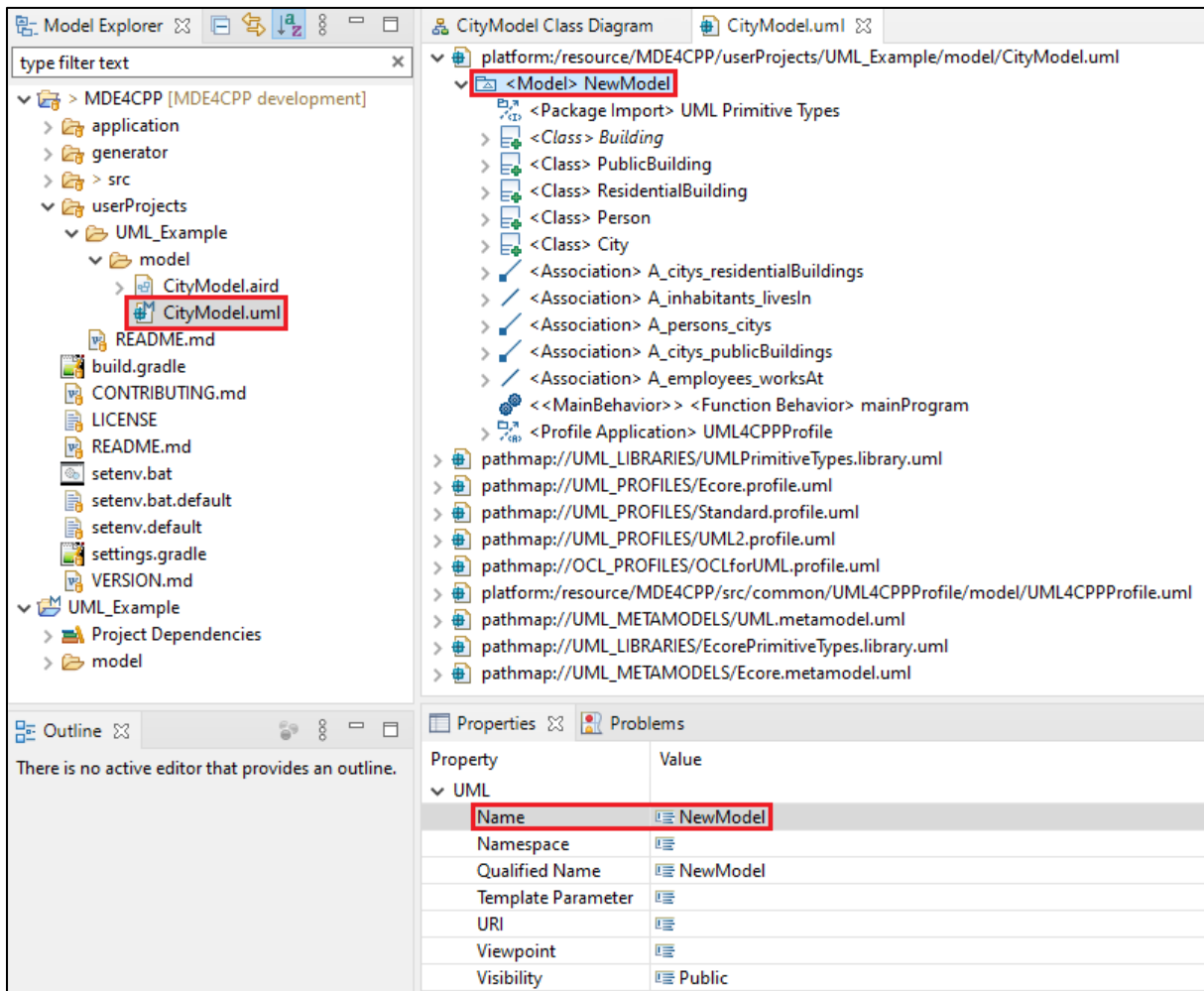
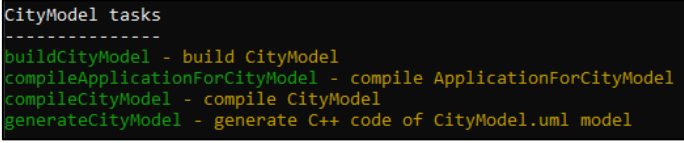


Figure 45: UML file name and model name do not match

To generate the finished model, navigate to the root directory of your MDE4CPP installation and open the window command line in this directory, as it was explained in “2.1 Starting eclipse via Command Line” and shown in Figure 2. Type “setenv.bat” and press Enter to temporarily set the environment variables. After that, type

```
gradlew generateModel -PModel="%MDE4CPP_HOME%\userProjects\UML_Example\model\CityModel.uml" -PSO
```

Press Enter and wait until the task is finished. Then, you can check whether the generation has worked by typing “gradlew tasks” and pressing Enter. After a couple of seconds, all available tasks will be shown. If you have set the model’s name to “CityModel”, scroll up until you see “CityModel tasks” (all tasks are sorted alphabetically). There should be 3 or 4 tasks as shown in Figure 46, depending on whether you added a main function via the MainBehavior stereotype.



```
CityModel tasks
-----
buildCityModel - build CityModel
compileApplicationForCityModel - compile ApplicationForCityModel
compileCityModel - compile CityModel
generateCityModel - generate C++ code of CityModel.uml model
```

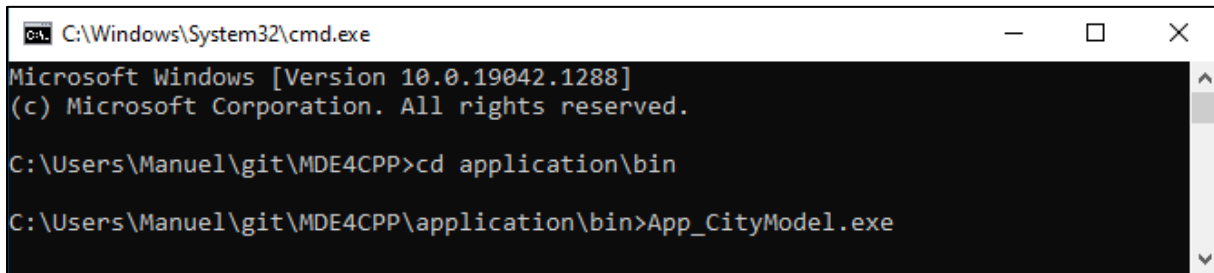
Figure 46: gradle tasks for the model

The generated program can then be compiled via “gradlew compile<Name>”, where <Name> is the model’s name starting with a capital letter. In the case of the example model, you must enter “gradlew compileCityModel”. This may take a minute. If everything is set up correctly, no errors should appear, and the task should end with a green “BUILD SUCCESSFUL” message. If you make changes to the model later, you can generate and compile the model in one step via “gradlew build<Name>”, where <Name> again is the model’s name starting with a capital letter. In the case of the example model, you must enter “gradlew buildCityModel”. If everything is set up correctly, no errors should appear, and the task should end with a green “BUILD SUCCESSFUL” message.

3 Running the application

To create a runnable application, the gradle task “compileApplicationForCityModel” is needed. Start it by entering “gradlew compileApplicationForCityModel”. This may take some seconds.

After the task is finished, you can start the generated application. Open a new Command Line Window and navigate to the MDE4CPP installation folder. Just like in “2.1 Starting eclipse via Command Line”, this can be achieved by pressing the Windows key and the R key at the same time (then the Run window opens), entering “cmd”, then clicking “OK” and entering “cd <path-to-MDE4CPP>”, where <path-to-MDE4CPP> is the path to the installation directory. This sets the workspace of the command line to the MDE4CPP home directory. Another way is to open Windows Explorer, navigating to the MDE4CPP installation directory, clicking on the directory path (marked in Figure 2), and typing “cmd”, then pressing Enter. This will also open the Command Line Window in the installation directory. Navigate into the bin-directory of the application folder by typing “cd application\bin” and pressing enter. To start the application, simply type “App_CityModel.exe” and press enter.



```

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19042.1288]
(c) Microsoft Corporation. All rights reserved.

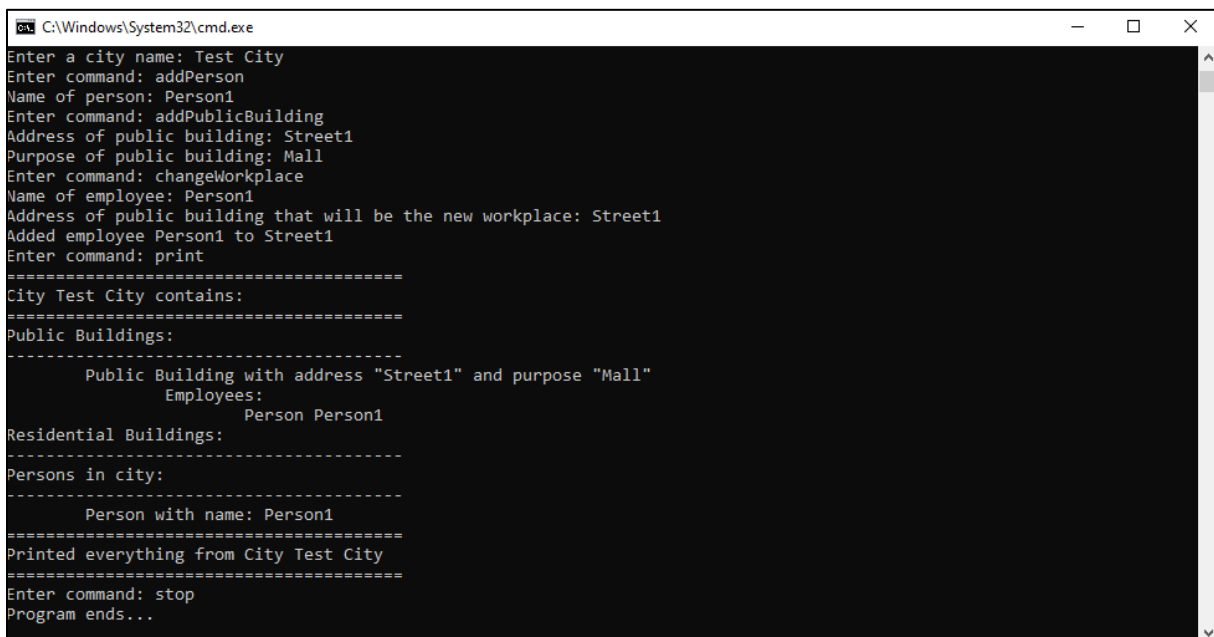
C:\Users\Manuel\git\MDE4CPP>cd application\bin

C:\Users\Manuel\git\MDE4CPP\application\bin>App_CityModel.exe

```

Figure 47: starting model application

After entering a city name and pressing Enter, you can type any command from the list of implemented commands from Code part 11 to Code part 17. Figure 48 shows a short example on how to use the commands. Start by typing “addPerson” and pressing Enter. This will create an object of the type “Person” in the city. In the next line (starting with “Name of person: “), you can enter the desired name. After pressing Enter, the input string is set as the person’s name. Typing “addPublicBuilding” and pressing Enter creates an object of type “PublicBuilding” as part of the city. In the following lines you can come up with an address and a purpose for the building. To assign the formerly created person to the building as employee, simply enter “changeWorkplace”. In the next two lines type in the name of the person and the address of the just created building. Afterwards, the command “print” can be used to output all existing objects in the city to the console. As you can see in Figure 48, there are one building with the address “Street1” and a person with the name “Person1” in the city. You can end the program by typing “stop” and pressing Enter. All available commands along with a short description can be viewed by typing “help”.



```

C:\Windows\System32\cmd.exe
Enter a city name: Test City
Enter command: addPerson
Name of person: Person1
Enter command: addPublicBuilding
Address of public building: Street1
Purpose of public building: Mall
Enter command: changeWorkplace
Name of employee: Person1
Address of public building that will be the new workplace: Street1
Added employee Person1 to Street1
Enter command: print
=====
City Test City contains:
=====
Public Buildings:
-----
      Public Building with address "Street1" and purpose "Mall"
      Employees:
        Person Person1
Residential Buildings:
-----
Persons in city:
-----
      Person with name: Person1
=====
Printed everything from City Test City
=====
Enter command: stop
Program ends...

```

Figure 48: Example how to use the application

This concludes the introduction to creating UML class diagrams and deriving code from them using MDE4CPP. With the tools explained in this document, simple to advanced C++ programs can be created independently.

4 Outlook and in-depth literature

As mentioned at the beginning, the goal was to explain the creation of a UML class diagram and the generation of the code from the class diagram. Complex issues (such as behavior-based models) were avoided, and the model was kept as simple as possible in order to convey the basics. Accordingly, work on these complex topics is conceivable in the future. In addition, the MDE4CPP program is constantly being further developed so that activity diagrams will also be supported in the future, which also opens up new opportunities for deepening knowledge in this area.

4.1 Activity diagrams

As part of the UML, activity diagrams are used to model both computational and organizational workflows, as well as the data flows intersecting with the related activities. Activity diagrams can also be created with eclipse by selecting “Activity Diagram” instead of “Class Diagram” in chapter 2.3, but code cannot be generated from them because that is not yet supported by the MDE4CPP generator. As already mentioned, this is in progress and should be possible in the future.

4.2 fUML

The fUML is a subset of the UML (including its typical structural modeling constructs such as classes, associations, or data types) but it also adds the ability to model behavior using UML activities composed of primitive actions. This leads to models constructed in fUML being executable in the same sense as a program written in a traditional programming language, with the difference that the fUML model is written with the level of abstraction of a modeling language.¹ Code generation for fUML models is already supported by MDE4CPP.

4.3 PSCS

PSCS is an extension of the fUML syntax and semantics and is short for “Precise Semantics of UML Composite Structures”. PSCS allows for modeling and execution of UML composite structures.²

4.4 OCL

Another approach to work in the future is the Object Constraint Language (OCL) that is a textual sublanguage of the UML. It can be used to express additional constraints on UML models that cannot be expressed, or are very difficult to express, with the graphical means provided by the UML. OCL is based on first-order predicate logic, but it uses a syntax similar to programming languages and closely related to the syntax of UML. It is, thus, more adequate for every-day modelling than pure first-order predicate logic.³ For both PSCS and OCL there are already some examples in MDE4CPP (under `src/ocl` or `src/pscs`), which can be used to understand how OCL and PSCS work.

¹ Definition is taken from <https://modeldriven.github.io/fUML-Reference-Implementation/>, slightly modified

² Definition is taken from <https://www.omg.org/spec/PSCS/1.2/About-PSCS/>, slightly modified

³ Definition is taken from <http://www-st.inf.tu-dresden.de/ocl/>, slightly modified