**NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES**

**Object Oriented Analysis & Design (CS-309)**

**Muhammad Nadeem || Awais Ahmed || Tooba Ali**

[**Muhammad.nadeem@nu.edu.pk**](mailto:Muhammad.nadeem@nu.edu.pk) **||** [**awais.ahmed@nu.edu.pk**](mailto:awais.ahmed@nu.edu.pk) **||** [**tooba.ali@nu.edu.pk**](mailto:majid.hussain@nu.edu.pk)****

**Lab Session # 01**

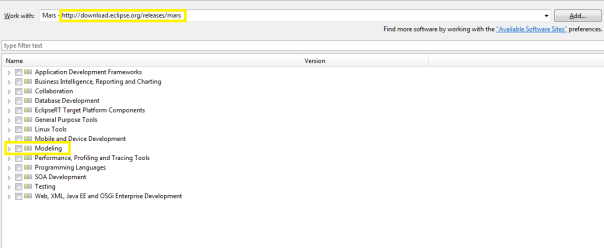
**Objectives:**

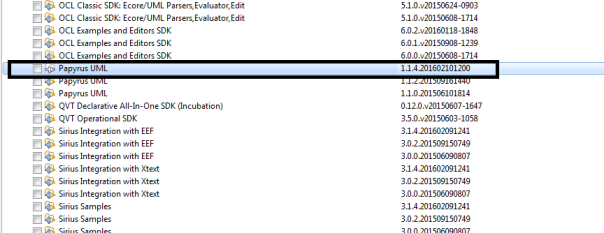
1. **How to interact with papyrus**
2. **Java profile and library**
   1. [**Java profile**](https://wiki.eclipse.org/Java_Code_Generation#Java_profile)
   2. [**Java library**](https://wiki.eclipse.org/Java_Code_Generation#Java_library)
   3. [**Common code generation profile**](https://wiki.eclipse.org/Java_Code_Generation#Common_code_generation_profile)
3. **Code generation from UML elements**
4. **Up model introduction**

How to install Papyrus and generate java code in Eclipse

Here, following steps you can download papyrus in existing eclipse and generate Java code using Class diagram.

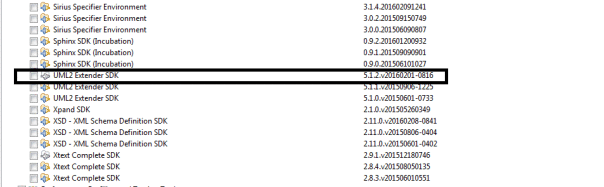
1. Install Papyrus plugin  
   Goto **Help** –>**Install New Software**  
   And in Work With enter URL  : <http://download.eclipse.org/releases/mars> <http://www.eclipse.org/oxygen/>



1. As shown in above image expand modeling node and look for **Papyrus UML.**Choose the latest version and select next and then finish by accepting eclipse license agreement**.**Papyrus UML Plugin

After installation it will ask for restart eclipse. Do restart.

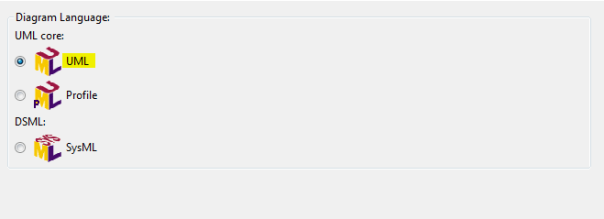
1. Install UML 2 for java code generation using papyrus.  
   UML 2 is not part of Papyrus plugin. So you need to install it separately as following.  
   Goto **Help –> Install New Software. Use**<http://download.eclipse.org/releases/mars>and in modeling node search for UML 2.



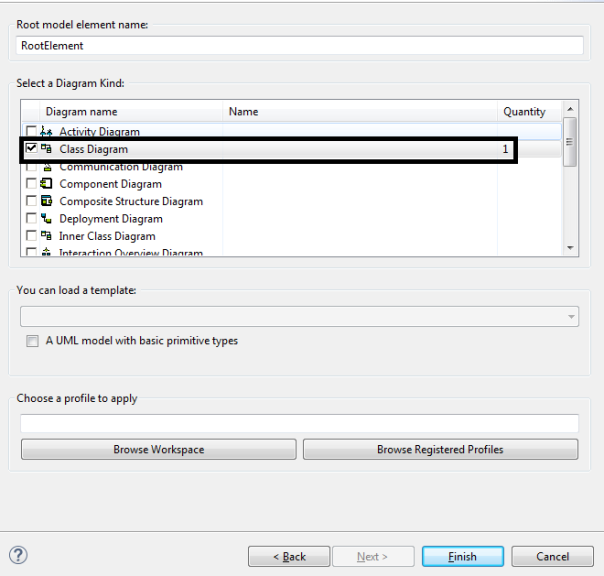
Uml 2 Plugin

Choose the latest version and select next and then finish by accepting eclipse license agreement**.**That’s it. We are done with installation.

1. Exercise : Java Code generation from papyrus.  
   Create one empty java project.  
   Right click on project and select **New –> Other.** Then select **Papyrus Model.**Select **Next.**In diagram Language Select **UML.**

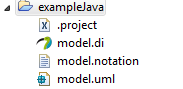
Select UML Diagram

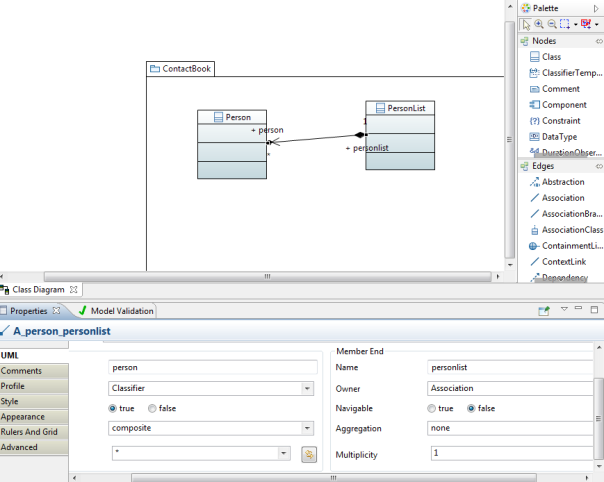
Then Select Next and select the Project name which is created as said in step 4.  
Then Select Next and select Class Diagram as below



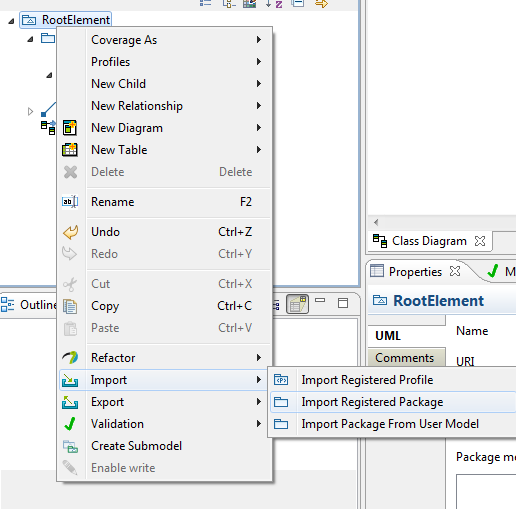
Class Diagram

Then Finish. it will generate all necessary files under your project.

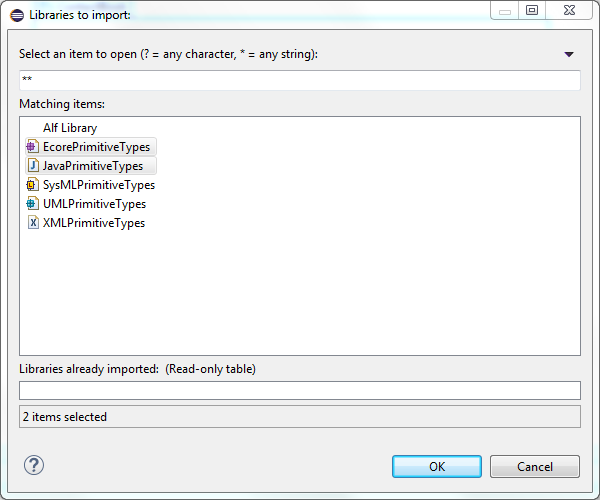
Example project

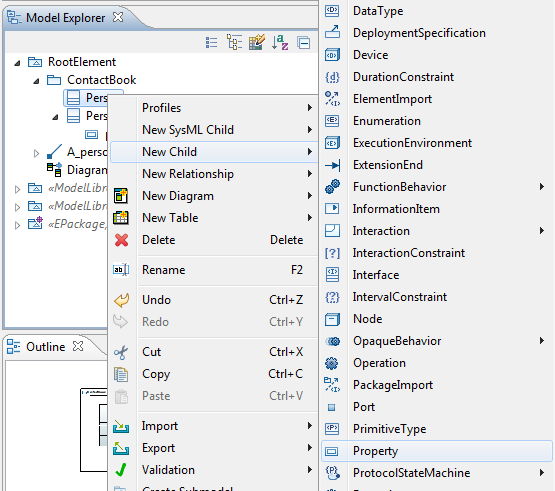
1. Open model.di file and change your perspective to **Papyrus** Perspective.
2. Create Class Diagram and attributes in it.Class diagram

Now, Import primitive types of J**ava** and **Emf** to use it for properties in class diagram as below.

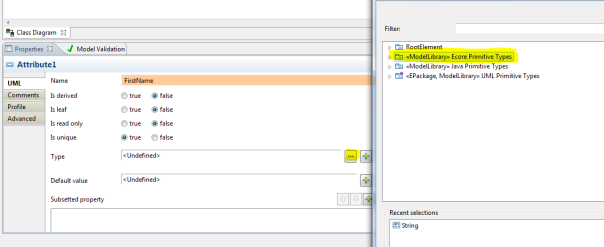
Import Registered Package

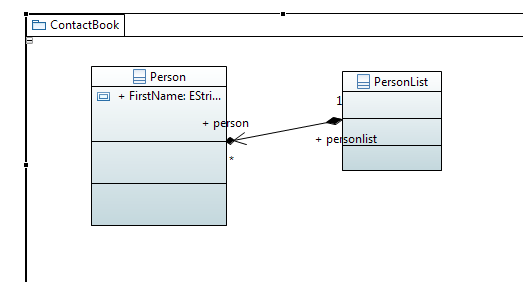
Select Ecore and Java Premitive Types as shown as below and select ok.

**Ecore** And **Java** Premitive types

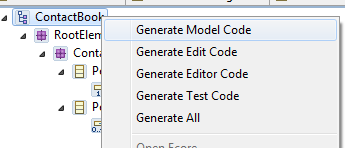
1. Go to Model Explorer and add right click on Person Class and add select **New child** –> **Property**
2. Add property in  class

Give Name and select Type String as shown as below in Ecore Premitive type.

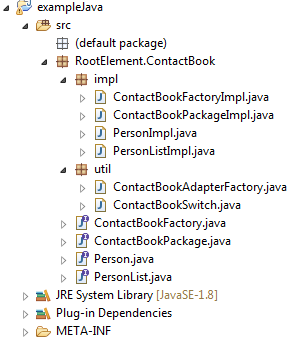
Assign Type to attribute

Drag and drop that property from Model Explorer to class in editor.  


1. Switch to Plugin development perspective and in project create new **gen model** as following.  
   Select project and open create new Wizard(**Ctrl + N**). Select **EMF Generator Model.**Give File name **ContactBook.genmodel.**Then select**UML model.** (Note if you don't find UML Model then UML 2 plugin is not installed properly).  
   Select Model URI by browsing workspace and select model.uml file in you project. Select Next and then **Finish**.
2. Generate Code for Contact Book as below.  
   Open **ContactBook.genmodel**and right click on Contact book and Select **Generate Model Code.**



Generate Model Code

1. **Thats IT!**It will generate code for you and convert your project in to plugin too.

Java Code Generation

The Java code generation is available in the extra plugins of Papyrus. It can be used by installing the latest version of Papyrus. It allows to create Java source code from a class diagram.

* [**1 Installation instructions**](https://wiki.eclipse.org/Java_Code_Generation#Installation_instructions)
* [**2 Code generation from UML elements**](https://wiki.eclipse.org/Java_Code_Generation#Code_generation_from_UML_elements)
* [**3 Generation preferences**](https://wiki.eclipse.org/Java_Code_Generation#Generation_preferences)
* [**4 Java profile and library**](https://wiki.eclipse.org/Java_Code_Generation#Java_profile_and_library)
  + [**4.1 Java profile**](https://wiki.eclipse.org/Java_Code_Generation#Java_profile)
  + [**4.2 Java library**](https://wiki.eclipse.org/Java_Code_Generation#Java_library)
  + [**4.3 Common code generation profile**](https://wiki.eclipse.org/Java_Code_Generation#Common_code_generation_profile)
* [**5 Incremental code generation**](https://wiki.eclipse.org/Java_Code_Generation#Incremental_code_generation)
* **6 Developer resources**

## Installation instructions

The Java features of Papyrus are available as part of the [**Papyrus Software Designer**](http://wiki.eclipse.org/Papyrus_Software_Designer) extra feature.

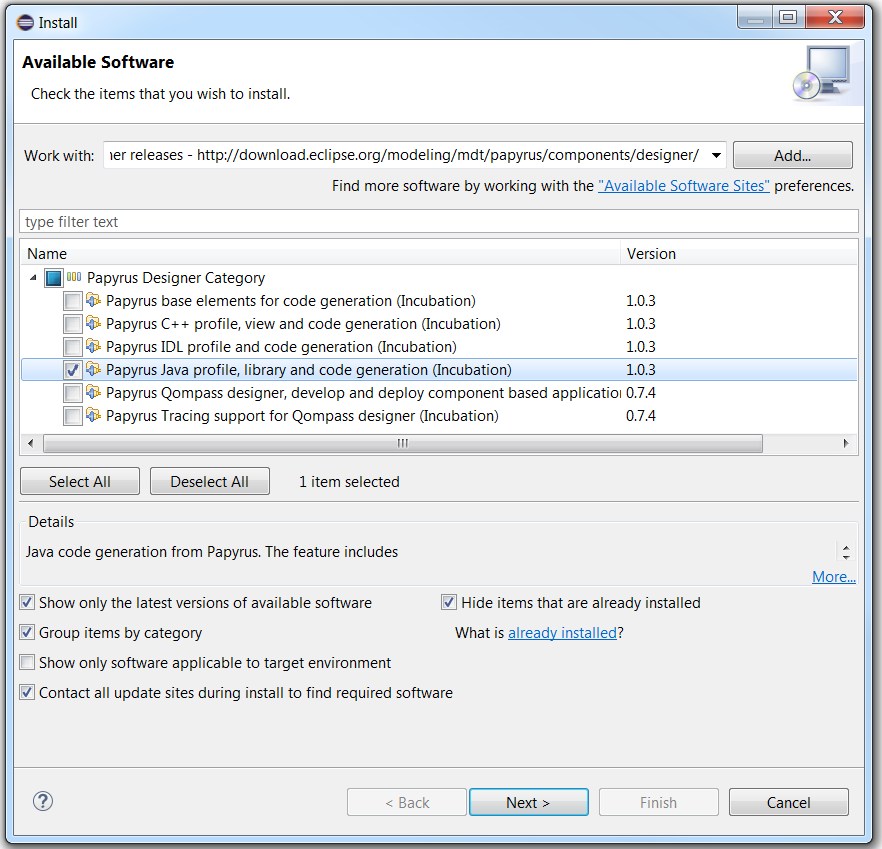
You can install it by using the Papyrus Software Designer update-site:

* [**http://download.eclipse.org/modeling/mdt/papyrus/components/designer/**](http://download.eclipse.org/modeling/mdt/papyrus/components/designer/)

Or you can use the nightly build update site (paste following url in eclipse update site):

* [**https://hudson.eclipse.org/papyrus/view/Designer/job/papyrus-designer-neon-papyrusnightly/lastSuccessfulBuild/artifact/releng/org.eclipse.papyrus.designer.p2/target/repository/**](https://hudson.eclipse.org/papyrus/view/Designer/job/papyrus-designer-neon-papyrusnightly/lastSuccessfulBuild/artifact/releng/org.eclipse.papyrus.designer.p2/target/repository/)

Then select "Papyrus Java profile, library and code generation (Incubation)" feature. You need JDT installed (which comes with most Eclipse packages).

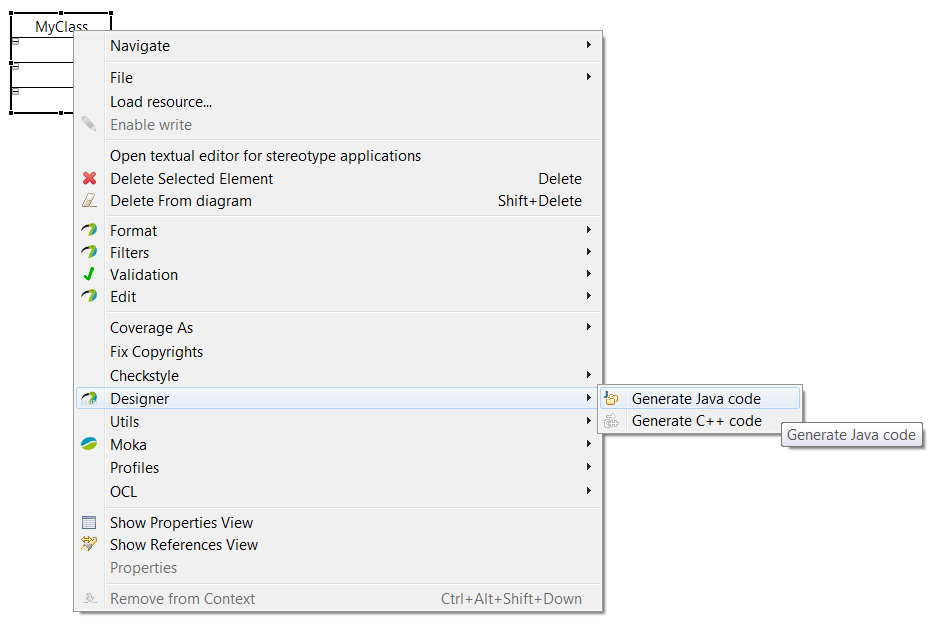
**[](https://wiki.eclipse.org/File:InstallSWDesignerJavaFeature.png)**

## Code generation from UML elements

You can generate code for a specific classifier or package in your UML model. To generate code, do the following:

* Right click on classifier or package, either in a diagram or in the model explorer
* Designer > Generate Java code
* Follow the JDT dialogs that let you create a new JDT project in Eclipse, where code will be generated, if there is no JDT project associated with your model (e.g. the first time you generate code from your model)

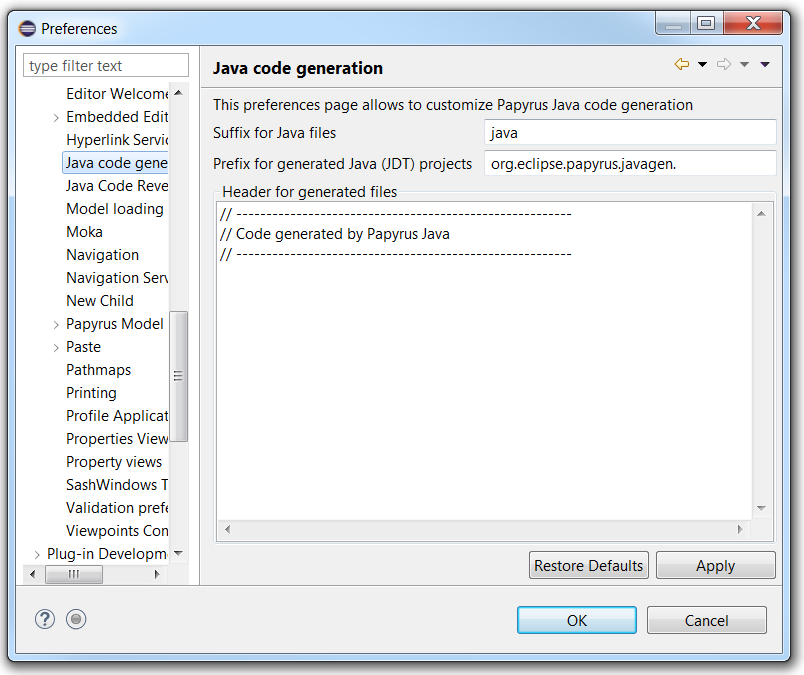
When you generate code from a classifier, its required classifiers are also generated. Required classifiers are classifiers related for the generated classifier, e.g. typing one of its attributes, inheritance relationship, dependency relationship. When you generate code from a package (e.g. the root of your model), all of its classifiers, and their required classifiers, will be generated.

**[](https://wiki.eclipse.org/File:GenerateJavaCode.jpg)**

## Generation preferences

You can change the Java code generation preferences using the Eclipse preferences menu, and choose Papyrus > Java code generation. Available options are:

* Extension for Java files
* Prefix for name of created JDT project (if unspecified in the model, see below)
* Header for generated files

**[](https://wiki.eclipse.org/File:JavaCodegenPrefs.png)**

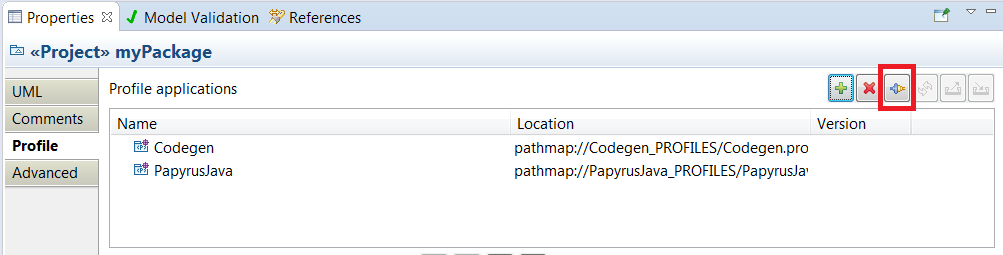
## Java profile and library

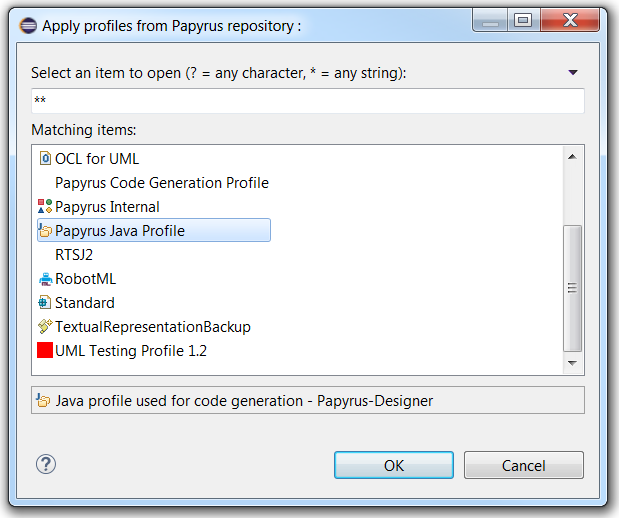
The installed Java feature, of Papyrus Software Designer, comes with a Java profile and a Java library. Furthermore, it also installs a generic code generation profile for meta-information (e.g. project name modeling).

### Java profile

The Java profile adds stereotypes to model specific Java elements that otherwise cannot be modeled in UML. It also adds stereotypes for managing elements that you do not want to generate (e.g. <<NoCodeGen>>, <<External>>, <<ExternLibrary>> stereotypes). The stereotypes can be seen as tags to put on UML elements. To apply the Java profile:

* Click on your model (in the Model Explorer view)
* In the Properties view, choose the Profile tab
* Click on Apply registered profile
* In the dialog that opens, choose the Java profile

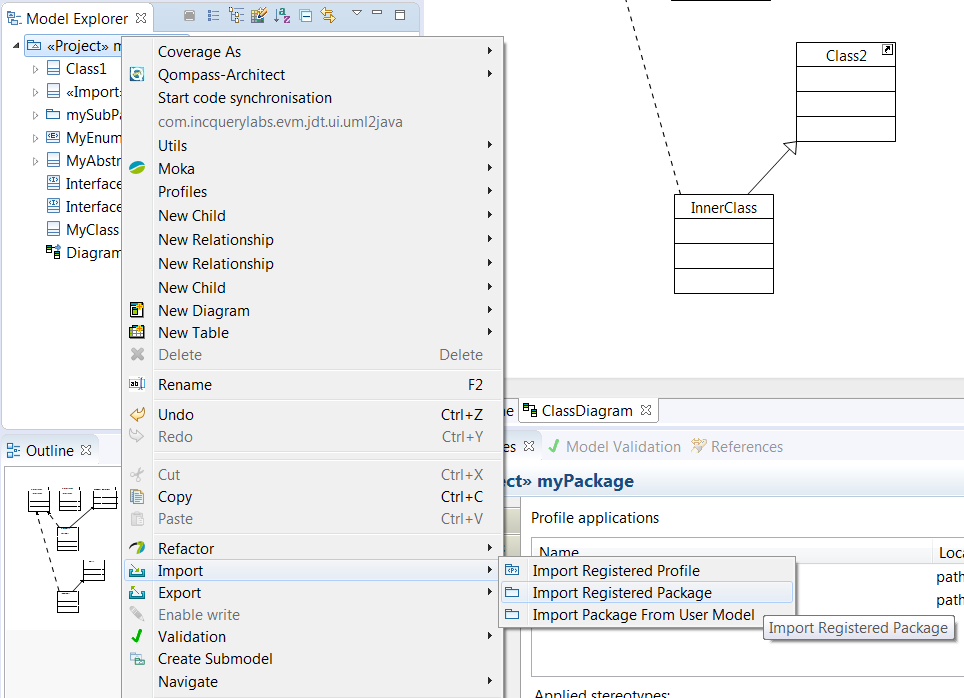
**[](https://wiki.eclipse.org/File:ApplyRegisteredProfile.jpg)**

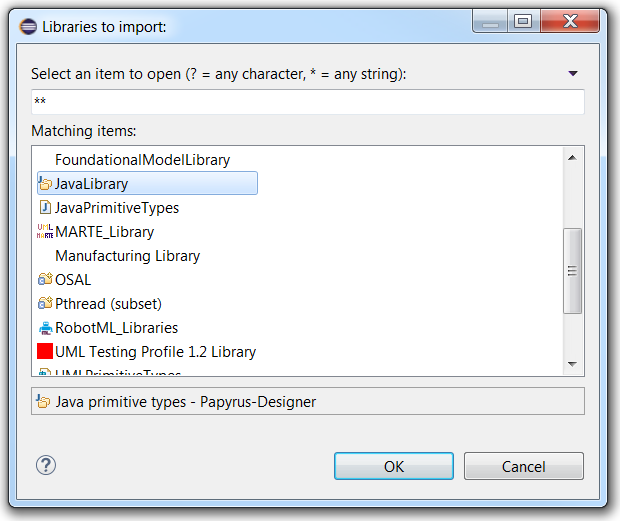
**[](https://wiki.eclipse.org/File:JavaRegisteredProfile.jpg)**

### Java library

The Java library provides Java primitives and their wrapper classes. To import, this library:

* Right click on your model (in the Model Explorer view)
* Import > Import Registered Package
* Choose the Java Library package
* Then you can use the Java primitives in this package, when you type attributes, parameters, etc...

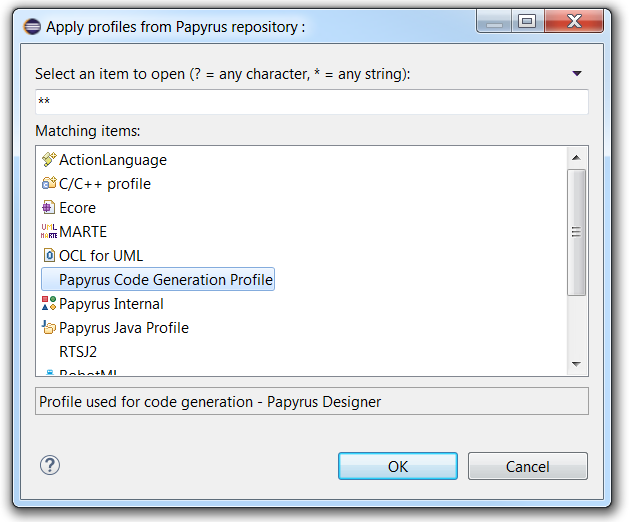
**[](https://wiki.eclipse.org/File:ImportPapyrusRegisteredPackage.png)**

**[](https://wiki.eclipse.org/File:JavaLibrary.png)**

### Common code generation profile

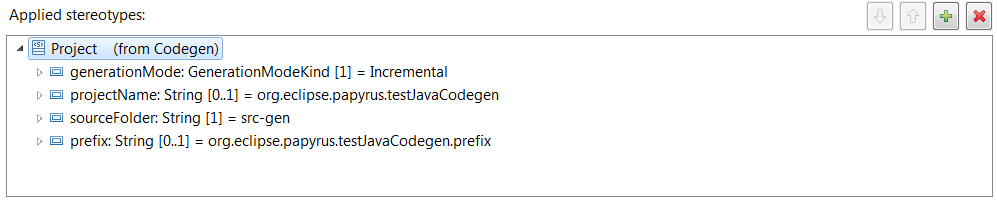
Papyrus Software Designer also offers a common code generation profile, that can be used for Java code generation. This profile lets the user model meta-information, e.g. path where code is generated.

To use this feature, first the Papyrus code generation profile must be applied:

**[](https://wiki.eclipse.org/File:CodeGenerationProfile.png)**

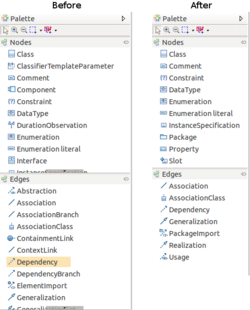
Then you can apply the <<Project>> stereotype to your model (root element). This stereotype lets you

* Generation mode: either batch or incremental
* Name of the JDT project (can be an existing one in the workspace)
* Source folder
* Prefix for all packages

**[](https://wiki.eclipse.org/File:SwDesignerProjectStereotype.png)**

## C:\Users\Majid.Hussain\Desktop\Untitled.png

The streamlined UML class diagram is assigned to the UML category and we base our implementation on the physical (hard-coded) UML class diagram in Papyrus (PapyrusUMLClassDiagram). In addition, we provide an optional [**custom style sheet**](https://wiki.eclipse.org/Papyrus_for_Information_Modeling/Customization_Guide#Styling) that is automatically applied to our UML class diagrams.

**[](https://wiki.eclipse.org/File:Papyrus_for_Information_Modeling_PapyrusCustomization_PaletteUI.png)**

*Palette Elements: Before and After*

**[](https://wiki.eclipse.org/File:Papyrus_for_Information_Modeling_PapyrusCustomization_ModelingAssistantsUI.png)**

*Modeling Assistants: Before and After*

Using the Papyrus UML class diagram as base, we need to constrain the elements available in the diagram to conform to our UML subset. In Papyrus, these constraints can be expressed as configuration rules. Each rule enables us to *allow* or *deny* certain behavior.

***Model rules***

Constrain the type of the model elements that can be visualized through the class diagram. In Information Modeling, we allow the visualization of UML Package elements, i.e., packages, models, and profiles.

***Owning rules***

Constrain the type of the model elements that can own the diagram itself. The owner is indicated as the element under which the diagram can be added in the Model Explorer. In Information Modeling, we match this behavior with the elements that can be visualized, i.e., we only allow packages to be owners.

***Child rules***

Constrain the type of the model elements that can be dropped within the diagram from the Model Explorer. In Information Modeling, we allow all modeled elements to be dropped into our class diagram.

***Palette rules***

Constrain the display of the diagram's palette elements. The palette is an area typcially shown at the right side of the editor and contains tools that allow the creation of new elements in the diagram. Palette elements are identified via their element id. In Information Modeling, we only display tools that allow the creation of elements that are part our UML subset. For instance

***Assistant rules***

Constraint the display of modeling assistants in the diagram editor. Modeling assistants are popup menus that are displayed when a user hovers with the mouse over an element in the diagram to support the creation of related elements, for instance, a property within a class. The elements in the menu are identified via their element type id. In Information Modeling, we only support the creation of elements that are part of our UML subset

Using these rules in the viewpoint configuration, we can customize Papyrus to a certain degree. Nevertheless, there are still aspects that can not be customized through the configuration model, such as the [**Creation Menus**](https://wiki.eclipse.org/Papyrus_for_Information_Modeling/Customization_Guide#Creation_Menu) and the [**Properties View**](https://wiki.eclipse.org/Papyrus_for_Information_Modeling/Customization_Guide#Properties_View).

*Class Diagram Example:*



Class Diagram Example

*Car.java*

[view source](http://java.dzone.com/articles/uml2-class-diagram-java#viewSource)

[print](http://java.dzone.com/articles/uml2-class-diagram-java#printSource)[?](http://java.dzone.com/articles/uml2-class-diagram-java#about)

01.public class Car {

02.private String carColor;

03.private double carPrice = 0.0;

04.public String getCarColor(String model) {

05.return carColor;

06.}

07.

08.public double getCarPrice(String model) {

09.return carPrice;

10.}

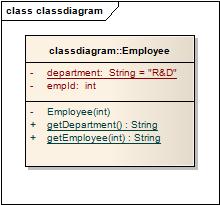
11.}

The above example of Car class is self explanatory. The Car class has private instance variables carColor, carPrice denoted by (-) in the UML Class diagram. Similarly if this was public then it would have been represented as (+), if was protected then it is denoted by (#). The package visibility is defined by (~).

|  |  |
| --- | --- |
| Java visibility | UML Notation |
| **public** | + |
| **private** | - |
| **Protected** | # |
| package | ~ |

The return type of the instance variables or the methods are represented next to the colon (:) sign.  
Structure:  
[visibility] [multiplicity] [:type [=default value]] {property string}  
Example: carPrice : double = 0.0

Representing Static variable or static operation:  
The static data is represented with an underline. Let’s take the below example.



Class diagram

Code:

[view source](http://java.dzone.com/articles/uml2-class-diagram-java#viewSource)

[print](http://java.dzone.com/articles/uml2-class-diagram-java#printSource)[?](http://java.dzone.com/articles/uml2-class-diagram-java#about)

01.public class Employee {

02.private static String department = "CS";

03.private int empId;

04.private Employee(int employeeId) {

05.this.empId = employeeId;

06.}

07.public static String getEmployee(int emplId) {

08.if (emplId == 1) {

09.return "ABC";

10.} else {

11.return "Employee not found";

12.}

13.}

14.public static String getDepartment() {

15.return department;

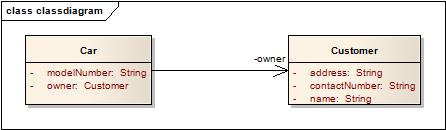
16.}

17.}

**Association:**  
The association represents the static relationship between two classes along with the multiplicity. E.g. an employee can have one primary address associated with it but can have multiple mobile numbers.  
Association are represented as thin line connecting two classes. Association can be unidirectional (shown by arrow at one end) or bidirectional (shown by arrow at both end).  
Multiplicity defines how many instances can be associated at any given moment.

|  |  |  |
| --- | --- | --- |
| 0..1 | No instances or one instance | A flight seat can have no or one passenger only |
| 1 | Exactly one instance | An order can have only one customer |
| 0..\* or \* | Zero or more instances | A class can have zero or more students. |
| 1..\* | One or more instances (at least one) | A flight can have one or more passenger |

The unidirectional relationship shows that the source object can invoke methods of the destination class. In Java a possible example can be the instance variable of source class referencing the destination class.



Association Example

[view source](http://java.dzone.com/articles/uml2-class-diagram-java#viewSource)

[print](http://java.dzone.com/articles/uml2-class-diagram-java#printSource)[?](http://java.dzone.com/articles/uml2-class-diagram-java#about)

01.public class Customer {

02.private String name;

03.private String address;

04.private String contactNumber;

05.}

06.

07.public class Car {

08.private String modelNumber;

09.private Customer owner;

10.}

Let’s look at an example of bidirectional association:



Bidirectional association

[view source](http://java.dzone.com/articles/uml2-class-diagram-java#viewSource)

[print](http://java.dzone.com/articles/uml2-class-diagram-java#printSource)[?](http://java.dzone.com/articles/uml2-class-diagram-java#about)

01.public class Customer {

02.private String name;

03.private String address;

04.private String contactNumber;

05.private Car car;

06.}

07.

08.public class Car {

09.private String modelNumber;

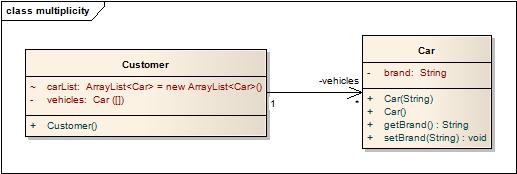
10.private Customer owner;

11.}

In the bidirectional association each of the class in this relationship refers to each other by calling each others method. In the above Java example it is depicted as instance variable of Car class in called inside the Customer class and vice versa.  
In the above example the car and owner refers to the roles and is depicted by the name of instance variable in the code.

**Multiplicity:**

Assume a scenario where a customer has multiple cars. How do we represent this situation in Java and UML?



Multiplicity in association

The above diagram explains a unidirectional association with a one to may relationship. Both use of ArrayList and Array is for illustration purposes only.

*Car.java*

[view source](http://java.dzone.com/articles/uml2-class-diagram-java#viewSource)

[print](http://java.dzone.com/articles/uml2-class-diagram-java#printSource)[?](http://java.dzone.com/articles/uml2-class-diagram-java#about)

01.public class Car {

02.private String brand;

03.

04.public Car(String brands){

05.this.brand = brands;

06.}

07.public Car() {

08.}

09.public String getBrand() {

10.return brand;

11.}

12.

13.public void setBrand(String brand) {

14.this.brand = brand;

15.}

16.

17.}

*Customer.java*

[view source](http://java.dzone.com/articles/uml2-class-diagram-java#viewSource)

[print](http://java.dzone.com/articles/uml2-class-diagram-java#printSource)[?](http://java.dzone.com/articles/uml2-class-diagram-java#about)

01.public class Customer {

02.private Car[] vehicles;

03.ArrayList<Car> carList = new ArrayList<Car>();

04.public Customer(){

05.vehicles = new Car[2];

06.vehicles[0] = new Car("Audi");

07.vehicles[1] = new Car("Mercedes");

08.

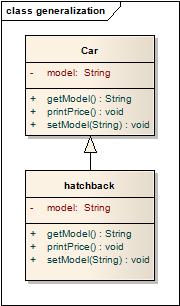
09.carList.add(new Car("BMW"));

10.carList.add(new Car("Chevy"));

11.}

12.}

**Generalization**  
This property represents the inheritance feature of the object oriented concept. In Java this can relate to the “extends” keyword. The inheritance should ideally follow the Liskov Substitution Principle i.e. the subtype should be able to substitute for its supertype. It helps to make the code implicitly follow the Open Close Principle i.e. Open for extension but closed for modification.

[](http://idiotechie.com/wp-content/uploads/2012/12/generalization.jpg)

Generalization

[view source](http://java.dzone.com/articles/uml2-class-diagram-java#viewSource)

[print](http://java.dzone.com/articles/uml2-class-diagram-java#printSource)[?](http://java.dzone.com/articles/uml2-class-diagram-java#about)

01.public class Car {

02.private String model;

03.public void printPrice() {

04.}

05.public String getModel() {

06.return model;

07.}

08.public void setModel(String model) {

09.this.model = model;

10.}

11.}

12.

13.public class hatchback extends Car {

14.private String model;

15.public void printPrice() {

16.System.out.println("Hatchback Price");

17.}

18.public String getModel() {

19.return model;

20.}

21.public void setModel(String model) {

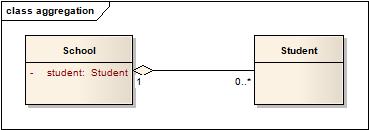
22.this.model = model;

23.}

24.}

**Aggregation:**

This shows “has a” relationship. It is a form of association relationship. This relationship highlights that a whole is made of its parts. So if a whole is destroyed the part still remains.  
In UML this is represented through a hollow diamond with the diamond symbol pointing towards the whole.  
In case of Java the aggregation follows the same structure as association. It is represented through the instance variables of a class.



Aggregation

[view source](http://java.dzone.com/articles/uml2-class-diagram-java#viewSource)

[print](http://java.dzone.com/articles/uml2-class-diagram-java#printSource)[?](http://java.dzone.com/articles/uml2-class-diagram-java#about)

1.public class Student {

2.

3.}

4.

5.public class School {

6.private Student student;

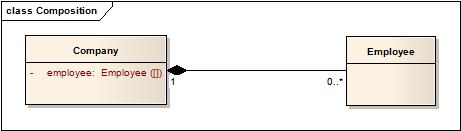
7.}

In this case a student is a part of the School. However during design it is preferred to use association instead of aggregation as it is not a recommended option.

**Composition**:

This is again a whole or part relationship where if the whole is destroyed then the part cannot exist independently. Another important point about Composition is that the part at any point in time can have only one owner. E.g. A person can be an employee of one company at any point in time due to contractual obligations. That person cannot hold dual work authorization. If the Company goes bankrupt the employee of this company does not exist and will be fired.

The composition is represented as a filled diamond with data flowing in single direction from the whole to the part.  
The composition in Java is represented in the same form as aggregation with help of instance variables.



Composition

[view source](http://java.dzone.com/articles/uml2-class-diagram-java#viewSource)

[print](http://java.dzone.com/articles/uml2-class-diagram-java#printSource)[?](http://java.dzone.com/articles/uml2-class-diagram-java#about)

1.public class Employee {

2.

3.}

4.

5.public class Company {

6.private Employee[] employee;

7.}