# Lab 06: UML

In this lab we will continue our work on diagramming by examining UML briefly, and class diagrams in detail. UML (the Unified Modelling Language) is designed to visualise software. As software has several ways of being viewed, UML likewise provides a number of diagram types. We will only mention four here but the [Wikipedia page on UML](https://en.wikipedia.org/wiki/Unified_Modeling_Language) points to further information.

## Behavioural Objectives

* [ ] **Define** the *two main types of UML diagram*.
* [ ] **Describe** the *elements of a UML class diagram*.
* [ ] **Add code** to an *application via a class diagram*.

## UML Diagram Types

UML diagrams can be divided into two broad types:

* **Behavioural diagrams**: those which describe the behaviour of the system as it executes.
* **Structural diagrams**: those which describe the static structure of the system.

The four most common diagram types are **use case diagrams**, **activity diagrams**, **class diagrams**, and **sequence diagrams**. Class diagrams are a *structural diagram* whereas the other three are *behavioural diagrams*.

### Use Case Diagram

We covered use case diagrams in the last lab. Essentially, use case diagrams try to capture the abstract behaviour of a system at specification time. They are a useful tool for communicating with stakeholders or providing a high-level overview of specification behaviour. However, they tend to have little direct mapping to actual code developed.

### Activity Diagram

Activity diagrams you are likely familiar with, although the name might be unusual. An activity diagram is just a flow chart:

By ​spanish Wikipedia user Gwaur, CC BY-SA 3.0, Link

Whereas the use case diagram captures functionality required from a user story point of view, the activity diagram allows steps to be defined in the process. This comes closer to actual code to be written, and can be refined to the point of actual code statements if need be, although that is very low-level. As a learner, activity diagrams can be very useful to help you map out how to code at a low-level. As you become more experienced, activity diagrams become more abstract and a communication tool between stakeholders.

### Class Diagram

Is the focus of the main part of the lab so we will cover this later.

### Sequence Diagram

Sequence diagrams map the communication between components (objects) as a system executes, and focuses on the method calls between objects. For example:

By Coupling\_loss\_graph.svg - File:CheckEmail.png, CC BY-SA 3.0, Link

Here, an object of type Computer has a method checkEmail called on it. The Computer then calls methods on a Server object such as newEmail. The point of the sequence diagram is that it captures actual object interactions. The sequences themselves are likely already captured in the activity diagram, but now we are using concrete object specifications to bring our solution to code.

## UML Class Diagram Overview

Of the four diagrams discussed, class diagrams are the most complex. Here is a simple example:

By No machine-readable author provided. Noodlez84 assumed (based on copyright claims). - No machine-readable source provided. Own work assumed (based on copyright claims)., Public Domain, Link

A class diagram attempts to capture the details of a class in a diagram. There are quite a few details, including:

* the class or interface.
* the name of the class.
* relationships between classes (e.g., inheritance).
* attributes of the class (i.e., class variables) including types.
* methods of the class including parameter and return types.
* the visibility levels of attributes and methods (i.e., *private*, *public*, *protected*).

Each of these points requires some form of visual metaphor:

* classes are represented by rectangles.
* the name is put at the top of the rectangle.
* relationships are shown via lines and arrows between rectangles.
* attributes are listed in a section under the name with types noted after them.
* methods are listed in a section under the attributes with types and parameters noted.
* visibility is marked before an item: + for public, - for private, and # for protected.

### Class Diagram Relationships

There are several different relationship types between classes. The point of mapping them out is to understand how classes are related so when we make changes we know the likely impact. Below is the common relationship types:

By Yanpas - https://commons.wikimedia.org/wiki/File:Uml\_classes\_en.svg, CC BY-SA 4.0, Link

* *association* is the most generic relationship type.
* *inheritance* means that a class is a specialisation of another class.
* *realization/implementation* is when a class implements an *interface*.
* *dependency* is a special form of association where changes to a class means the dependant class will likely have to change.
* *aggregation* is a special form of association denoting a *has-a* relationship. This is not considered a strong relationship in so far as the class does not own the associated object.
* *composition* is as aggregation but now the class **owns** the object. When an instance of the owning object is destroyed so are its components.

[Lecture 11](../../lectures/lecture11) goes into more detail about relationship types.

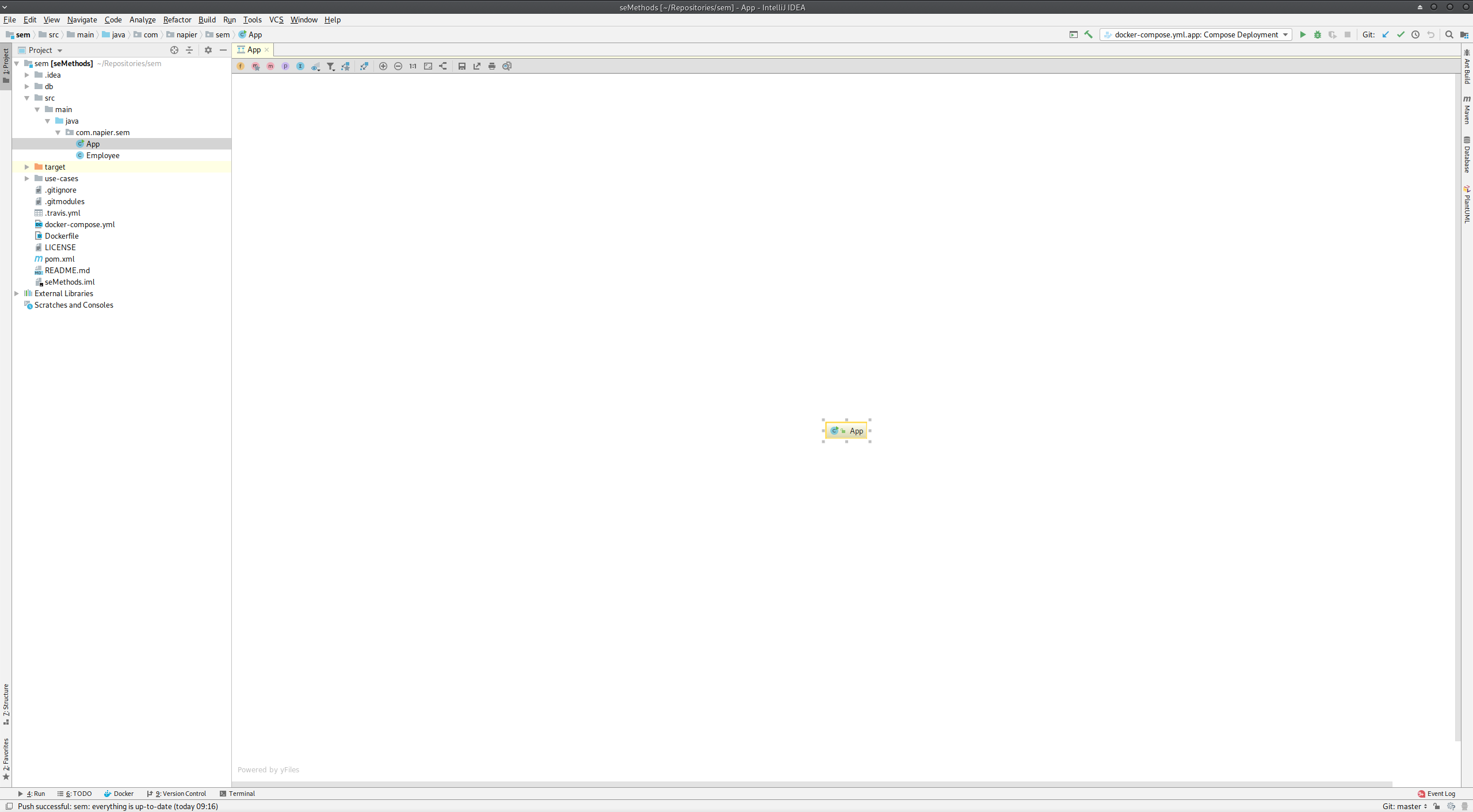
## Class Diagrams in IntelliJ

UML support used to be a key feature in an IDE but the requirements have changed as industry practice has moved. However, it is still common for an IDE to support class diagrams as they have a direct mapping to our code. IntelliJ is no different in that regard.

To use class diagrams in IntelliJ you need to ensure either:

* you have the Ultimate Edition installed; or
* you install the UML Support plugin.

Once ready, to view a class diagram simply **right-click** on a class (e.g., App) in the **Project View** and select **Diagrams, Show Diagram**. This should bring up the following view:



IntelliJ Diagram View

OK not exciting so far - we just have a box saying App on the screen. Let us add the details. At the top of the view you will see a row of buttons:

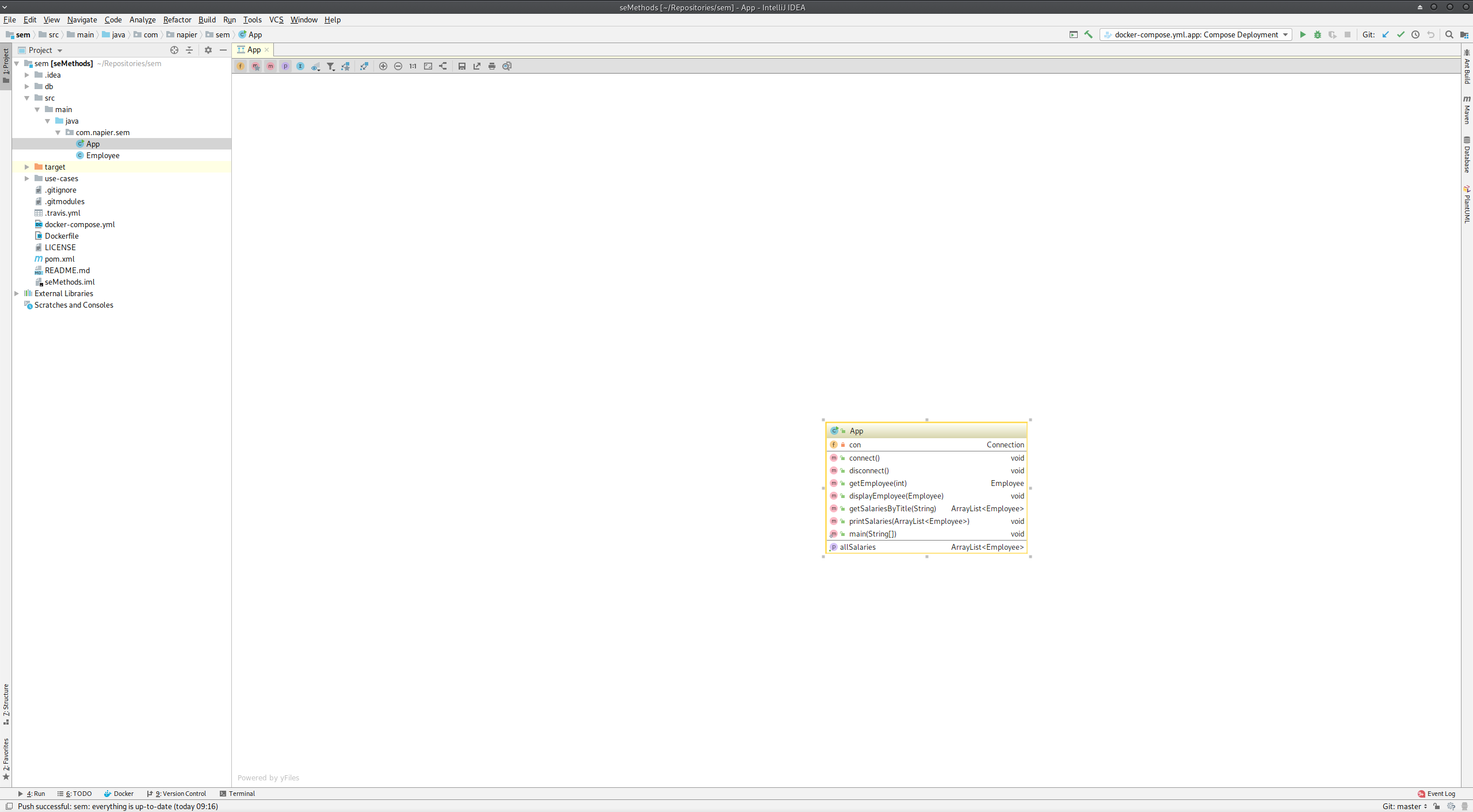
IntelliJ Diagram Buttons

IntelliJ Diagram Buttons

The first four are the ones we are interested in. These are:

* show fields: display the attributes.
* show constructors.
* show methods.
* show properties - the get and set style methods.

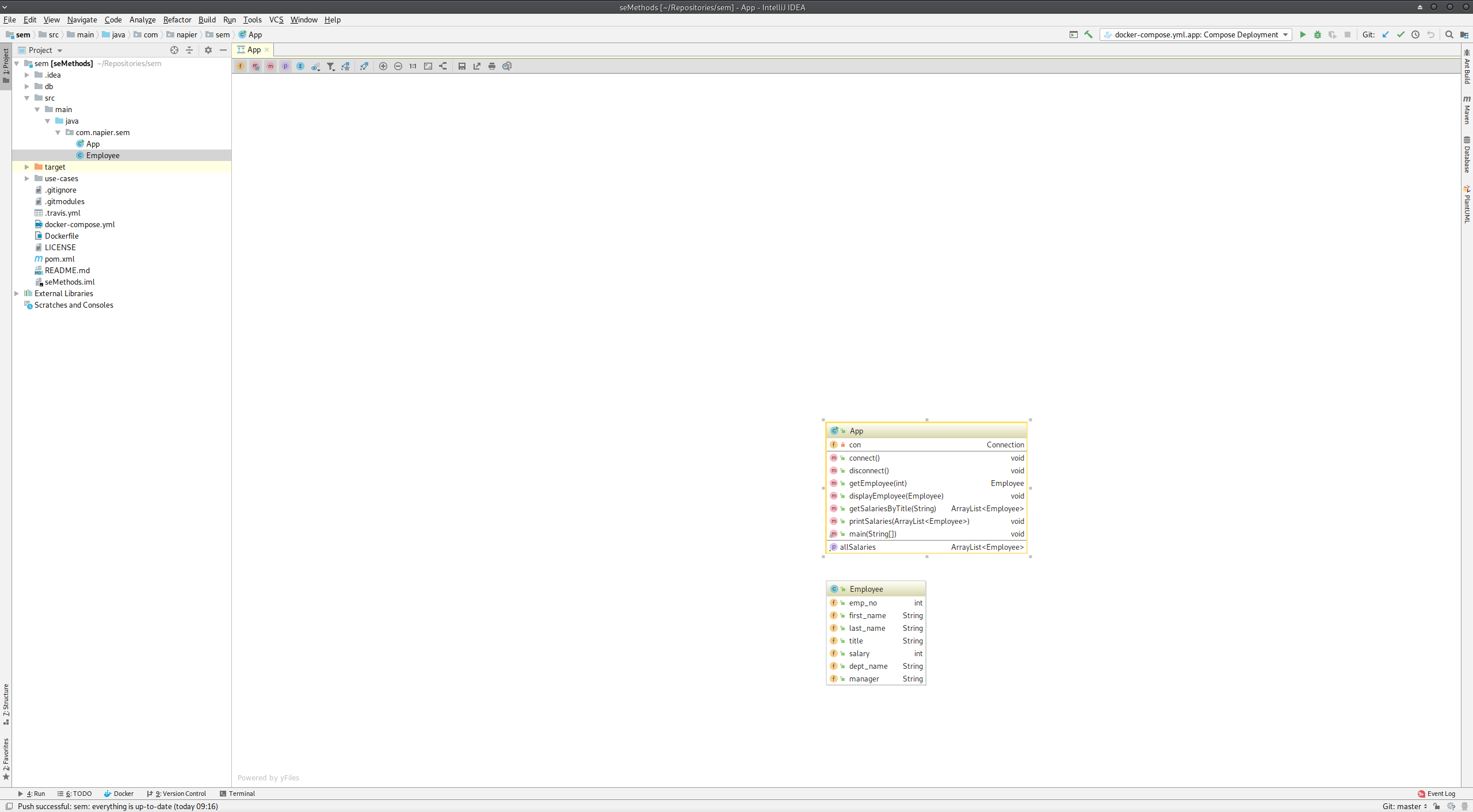
**Click these four buttons now**. Your window should now look like this:



IntelliJ Diagram with All Details

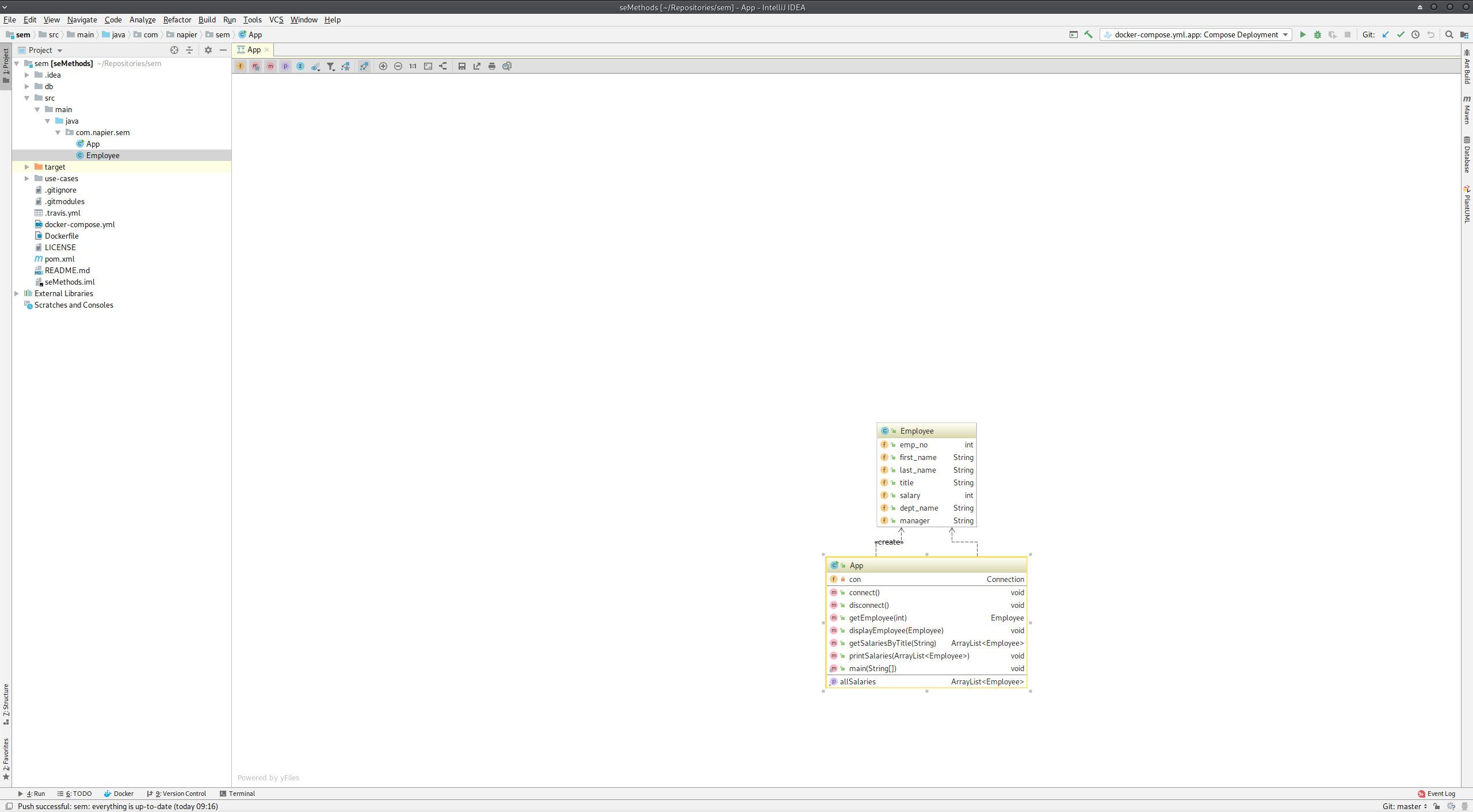
Note that IntelliJ does not use + or - do denote visibility but small padlocks. The information is otherwise the same.

**Now drag the Employee class from the Project View into the window.** This should display the details of the Employee class also:



IntelliJ Diagram with App and Employee

With two classes in the window, we can show the dependencies between the classes. We do this by **clicking the ninth button: Show Dependencies**. Our view now becomes:



IntelliJ Diagram with Dependencies

Notice that App has two dependencies on Employee: a standard one (since it is used as a parameter) and a create dependency. This is because App creates instances of Employee.

## Next Feature: Salary by Department

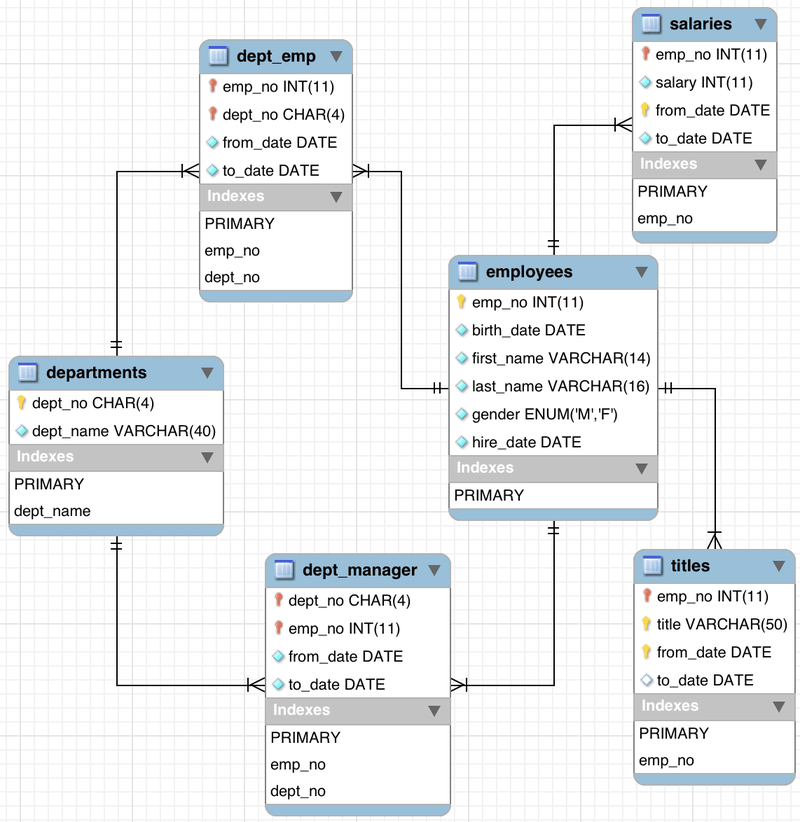
For our next feature we are going to build some of the functionality via the diagram view. We will still need to write logic code ourselves, but we can create some of the specification via the diagram view.

Our next feature is getting salary by department. This is similar to the feature last week.

**First, set yourself up to start this week’s Sprint.** Refer back to [Lab 04](../lab04) if you are unsure what to do.

### Add Department Class

Our feature requires a new class - Department. If we examine the [database schema](https://dev.mysql.com/doc/employee/en/sakila-structure.html) below we see that a department has two links to Employee - one as a collection of workers and another as a manager of a department.



Employees Database Schema

Our class diagram will make this evident as we progress. First, **add a new class Department to your diagram** by **right-clicking in the window** and selecting **New then Class**. Call the class Department. You should end up with the following:

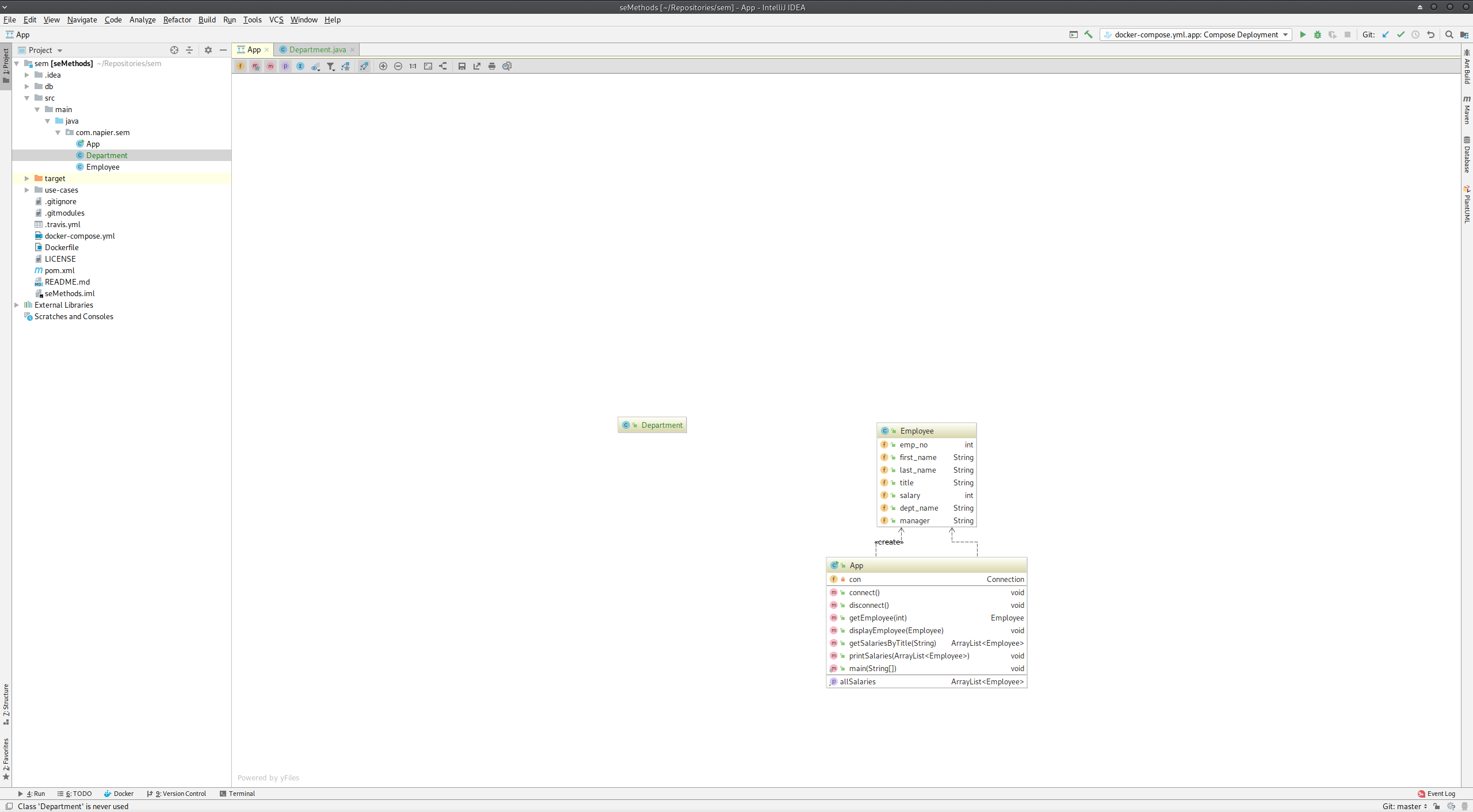
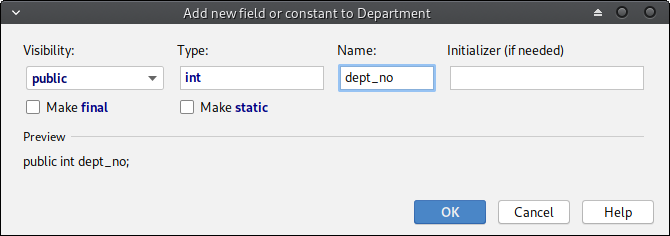


Diagram with Department added

A Department has the following information:

* the dept\_no (department number - is actually a String such as d001).
* the dept\_name (department name).
* the manager.

Our class will therefore need that information. We can do this in the diagram. **Right-click Department and select New then field**. This will bring up the **New Field Dialogue**:

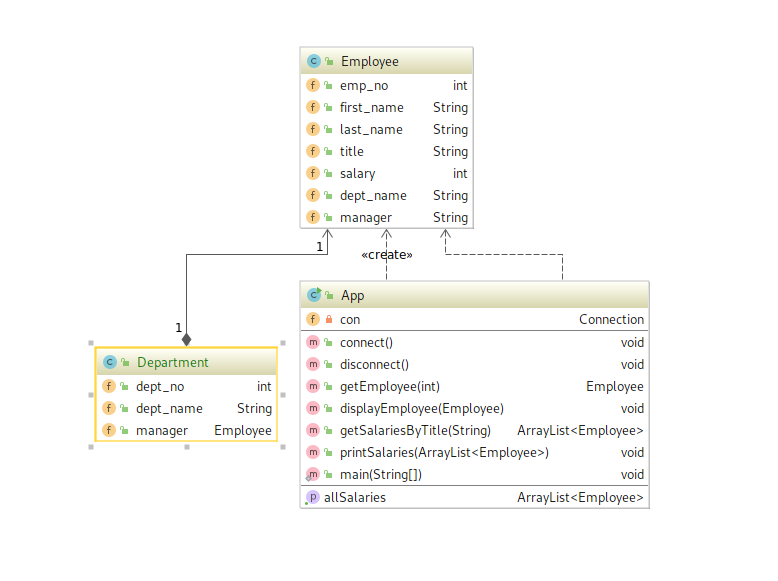


IntelliJ Diagram New Field

We need three fields:

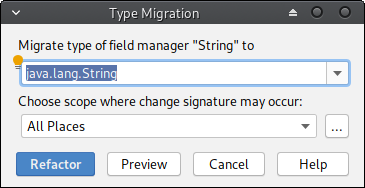
* public String dept\_no
* public String dept\_name
* public Employee manager

Add these now. Your class diagram should now look like this:



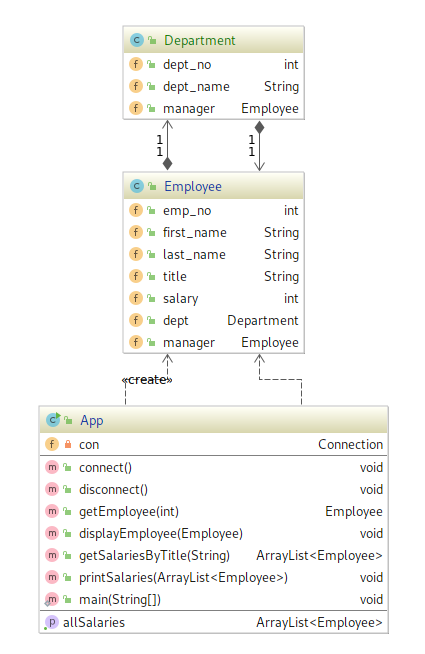
Class Diagram with Department Added

Note the automatic addition of a composition relationship. Also, we already had a manager attribute in Employee but it is a String. Let us change that to an Employee. **Double-click manager in Employee**, then **right-click it and select Refactor then Type Migration**. This will open the **Type Migration Window**:



IntelliJ Type Migration

**Change the type to Employee and click Refactor.** **Now do the same for dept\_name in Employee but change it to public Department dept.** To rename an attribute, use **Refactor, Rename**. Your updated diagram should look something like this:



Class Diagram after Refactor

We have now done a few basic things in our diagram:

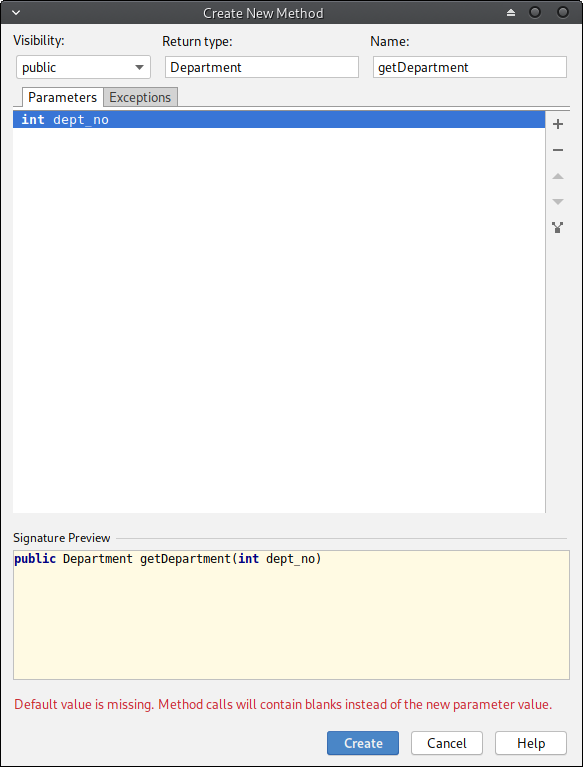
* added a new class.
* added fields to a class.
* modified a field in a class.

Let us now add methods to get department information.

### Add Get Department by Name Method

Once we have a department number or name we want to be able to create a department object. To do this, our application needs a new method called getDepartment. This will get a department object via its name.

To add a method to App, **right-click the class in the diagram and select New then Method.** This will open up the **New Method Window**:

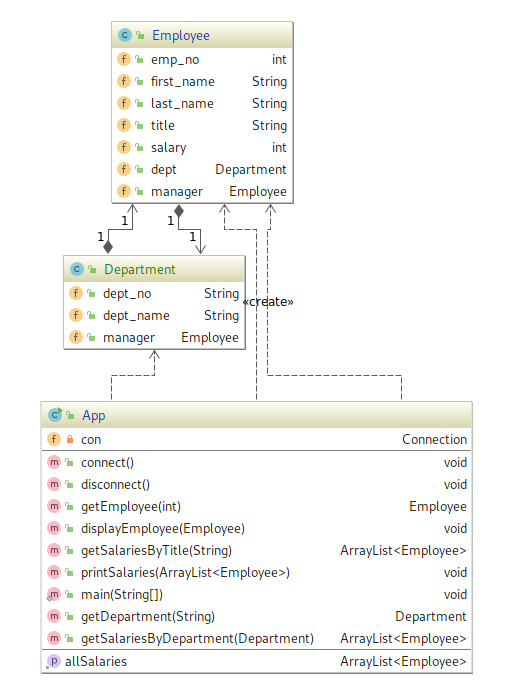


IntelliJ Diagram New Method

Adding a method is a little more complicated. You need to define the return type (Department), the name (getDepartment) and the parameters. To add a parameter, **click the little + sign on the right**. Then define the parameters as needed. The signature is:

* public Department getDepartment(String dept\_name)

After this, your class diagram should look like this:



Class Diagram with getDepartment

### Add Get Salaries by Department Method

To actually complete the feature we need a new method in App: getSalariesByDepartment. It has the signature:

public ArrayList<Employee> getSalariesByDepartment(Department dept)

**Add this method now.**

### Exercise: Implement the Feature

All the scaffolding is in place to finish the feature. You have to complete the following two methods:

* public Department getDepartment(String dept\_name)
* public ArrayList<Employee> getSalariesByDepartment(Department dept)

You will also need to update main to test the feature. **You will also need to update the comments to ensure your code is still well explained.** If you use getDepartment("Sales") for the department, your output should be as follows:

...  
499894 Maja Lamba 68787   
499895 Raimond Leuchs 98808   
499899 Mong Usdin 104333   
499901 Make Terekhov 49223   
499902 Aloke Wuwongse 100339   
499919 Masako Angiulli 107704   
499920 Christ Murtagh 123461   
499926 Youpyo Perfilyeva 109498   
499953 Leszek Pulkowski 114876   
499960 Gaetan Veldwijk 94157   
499966 Mihalis Crabtree 98388   
499976 Guozhong Felder 107386   
499980 Gino Usery 108364   
499986 Nathan Ranta 119906   
499987 Rimli Dusink 56336

If you need a hand, the SQL for the getSalariesByDepartment method is:

SELECT employees.emp\_no, employees.first\_name, employees.last\_name, salaries.salary  
FROM employees, salaries, dept\_emp, departments  
WHERE employees.emp\_no = salaries.emp\_no  
AND employees.emp\_no = dept\_emp.emp\_no  
AND dept\_emp.dept\_no = departments.dept\_no  
AND salaries.to\_date = '9999-01-01'  
AND departments.dept\_no = '<dept\_no>'  
ORDER BY employees.emp\_no ASC

### Clean-up and Commit

As before, you need to end the feature. Refer to [Lab 04](../lab04) for our current process.

## Exercises

These exercises require some work on the SQL mainly:

1. Add the department manager to the department in getDepartment if you have not done so already.
2. Update getEmployee to also add the department manager as an Employee.
3. Add a new method to getEmployee based on their first name and last name.