Exercise 6: Composition & Aggregation

Lab Objectives

- Start to look at relationships between classes, that can be specified in UML
- Understand how concepts of association, dependency and delegation can provide a basis for modeling software systems
- Design and implement classes associated by Composition and Aggregation, i.e. classes that encapsulate other objects

Associated Reading:

Horstmann,

Chapter 7, Sections 7.1 - 7.2: Designing Classes
Chapter 11, Sections 11.2 - 11.3: Object-Oriented Design
Appendix H, UML Diagrams: UML Summary

Associating multiple classes

So far you have learnt how to design and implement your own classes, as well as reusing existing classes to achieve simple tasks in test programs. As you start to make more complex programs you will need to work with multiple classes that will encapsulate different features of your system.

It is common practice to combine the functionality of different classes through a relationship called association - meaning two classes are associated with each other. When one class uses another class internally there is said to be a dependency. A class may depend on another class if it simply uses it to assist with a single task, e.g. during a method call.

There are two other notable types of association: composition and aggregation. These are both stronger forms of dependency as a class will hold objects of another class as one of its instance variables (i.e. fields). The most important difference between composition and aggregation that you should appreciate is that the former involves storing a single object of another class, whereas the latter is relevant when storing a collection of objects of another class.

They are sometime called *whole-part* relationships. The distinguishing feature of a composition is that the lifetime of the parts is the same as the whole container, whereas the objects in an aggregation can exist outside of the containing object. Typically an aggregation is used when the *whole* maintains a *collection of parts* of the same type, i.e. a "has-many" relationship, whereas composition is appropriate where the *whole* has a fixed number of instances of a particular class as a *part*, i.e. a "has-a" relationship (or "has-two", "has-three", etc).

From a class design point of view, there is little difference whether an association is a composition or an aggregation. The key aspect is the multiplicity, "has-a" or "has-many".

Two fundamental principles of object oriented systems are that of encapsulation and code reuse. Whenever a set of data or methods are used in more than one place, you should try to encapsulate it so you can reuse the code and isolate any changes you may need to make in the future. There is no point rewriting code to achieve a task that is already possible by reusing existing components - this is what associations are all about.

When classes use or store other objects they can often reuse methods belonging to those objects - this principle is called delegation.

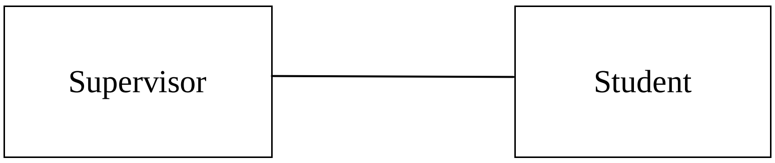
Specifying the relationships

The main type of relationships between two class are:

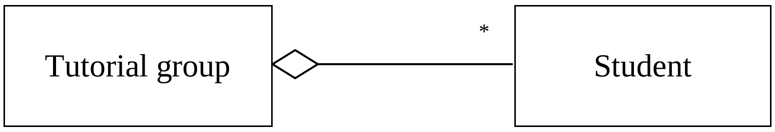
- *Composition* (building a whole from several different parts) **"has-a"**
- *Aggregation* (a collection of objects of the same type) **"has-many"**
- *Dependency* (uses another class) **"uses"**

These relationships can be illustrated in UML class diagrams. An association is shown by drawing a line between classes. The various arrow heads give the specific type of association. The lines can also be annotated with multiplicities, relationships and roles.

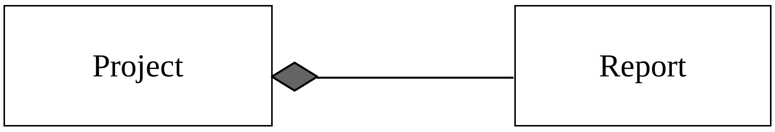
Examples:



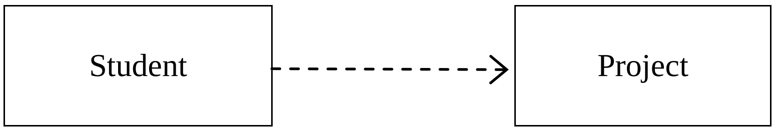
Two-way association - a supervisor supervises a student. A student has a supervisor.



Aggregation - usually a collection. A tutorial group enrolls several students.



Composition (whole – part). A project "**has a**" report.



Dependency ("**uses**") A student chooses a project.

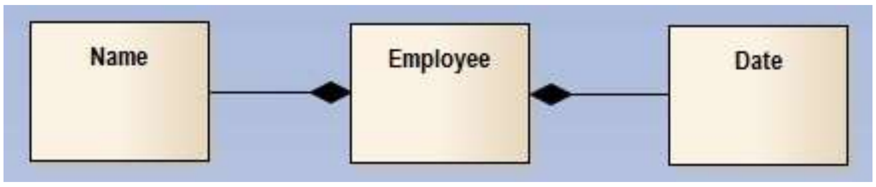
Sometimes the distinction between an aggregation and composition is not clear. The advice for now is not to worry too much about it. The relationship is still essentially an *association*. The distinguishing feature of a composition is that the lifetime of the parts is the same as the whole. E.g. a project report does not make sense without a project, whereas student exists independently of a tutorial group.

Composition

A class may have fields that are themselves objects. We say the object is composed of other objects. It is a "whole–part" relationship. This kind of relationship between classes may be shown in UML as a composition (i.e. a filled in diamond).

Technically speaking you have already implemented some classes that have utilized composition - classes you have built that have a field of type String. However, as String is such a commonly used type and already exists in the API, it is not generally specified in UML diagrams.

Question 6.1 Reopen your Eclipse workspace called "**CTEC2905_OO_Design**" then download the existing Java Project that can be extracted from the "**Composition.zip**" file, and imported into Eclipse. In Eclipse press Ctrl+F10 and select Package Presentation > Flat. In the lib.employee package you will find the **Employee** class that is composed of objects of type Name and Date, and also holds a salary.

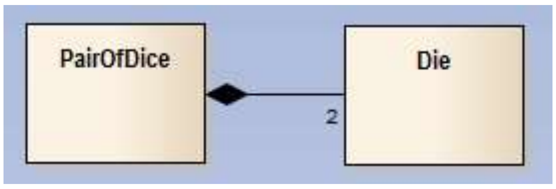


You have already seen the Name class, but may wish to briefly look at the Date class, then take a more detailed look over the **Employee** class to see how the composition relationships have been formed. You should see similarities with the general structure of this class when compared with others you have developed, with the main difference being that the types are of Name and Date, instead of String or primitive types, that you have become used to working with.

You may have noticed that each class overrides the equals(...) method, and that the Employee implementation delegates to the equals method in the Name and Date classes to help test the equality of those instance variables, whilst using == for the primitive type *salary* field is suffice.

Study the code within the **EmployeeDemo** program in the main package to see the different way in which Employee objects can be constructed and used via the methods it offers.

Question 6.2 In the lib.dice package, study the **PairOfDice** class that is composed of two Die objects. It implements the Rollable interface and therefore overrides the **roll** method by rolling both dice, and the **getScore** method by returning the sum of their scores. It achieves this through the principle of delegation - reusing the existing functionality of the Die class.



Take a look at the **RollableDemo** program - you will notice it is similar to the DieApp program from week 4, however, there are two variables of type Rollable, one assigned a PairOfDice, and another a Die. By using the interface type, we can interchange between these two classes.

Notes - Invoking multiple methods in a single statement:

- Up until now you have generally been invoking a single method on a single object instance, e.g.

```
Name n = new Name("Joe", "Bloggs");
System.out.println(n.getFirstName());
```

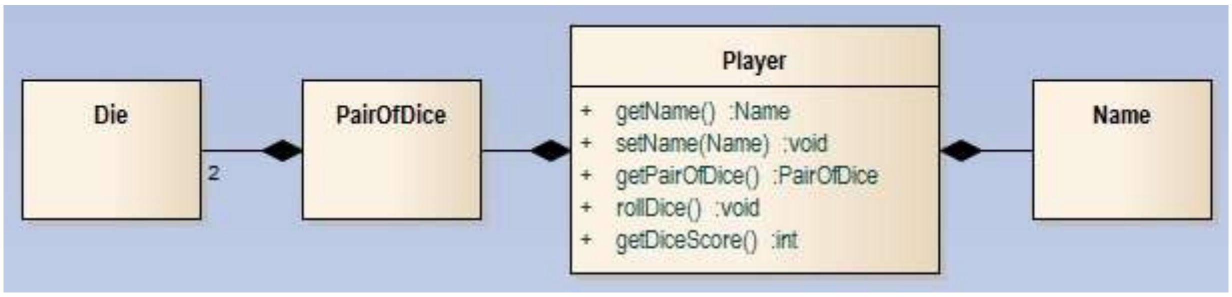
- As you start to create classes that are composed of other objects, you will find you may use the dot notation to invoke multiple methods in a single statement, e.g.

```
Employee emp = new Employee();
System.out.println(emp.getName().getFirstName());
emp.getName().setFirstName("Joe");
```

- In the code above the getName() method returns a Name object reference and therefore any of the methods that belong to the Name class can be invoked on that.
- There is no limit to the amount of methods that can be called in a single statement, take for example an ArrayList of type Employee:

```
ArrayList<Employee> employees = new ArrayList<Employee>();
employees.get(0).getStartDate().getSetDay(21);
```

Question 6.3 [✱Portfolio A.1✱] Download the existing Java Project that can be extracted from the "**StartingPlayer.zip**" file, and imported into Eclipse. This question forms the first part of Portfolio case study A. Your task is to create a **Player** class that is composed of a Name and PairOfDice. In next week's exercise, you will update the Player class, and be given some unit tests for the updated version.



a) The **Player** class should have a default constructor and two custom constructors - one that accepts a Name object, and another that accepts both a Name and PairOfDice object.

b) There should be get and set methods for its Name and a get method for PairOfDice. It should have a method called **rollDice** and **getDiceScore** that both simply delegate to the PairOfDice class, which already has this functionality. You should also have an appropriate **toString()** method.

c) Add a further void method **setFullPlayerName(String)** that accepts a single `String` argument (e.g. "Joe Bloggs") and then uses this to set the first and family name individually by extracting the relevant information and then calling the respective setter methods of the Name class.

d) As this is a portfolio question, you should fully document the Player class with Javadoc, i.e. class header, constructor/method description and tags where appropriate.

Notes - The principle of delegation:

- You have already seen how the PairOfDice class delegated to the Die class to roll both its dice and calculate their combined score - this is common practice when working with associations.
- When an aggregation relationship exists, the aggregate class (i.e. that which holds a collection) can take advantage of delegation by delegating responsibility to the internal data structure.
- For example the ArrayList class already has an add(...) method, so a PlayList class can simply delegate this task when adding a song to the internal list:

```
public void addSong(Song s) {
      songlist.add(s);

}
```

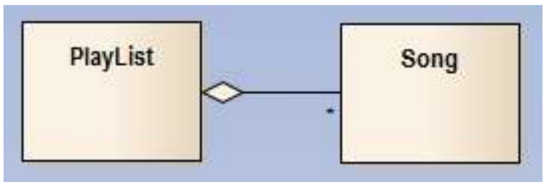
- You may wonder why you would build a class, where you are primarily delegating responsibility to another class. The benefit is you have created something new (in this case a PlayList), which only provides selected necessary functionality of an ArrayList and can additionally add functionality of its own.

Please turn over...

Aggregation

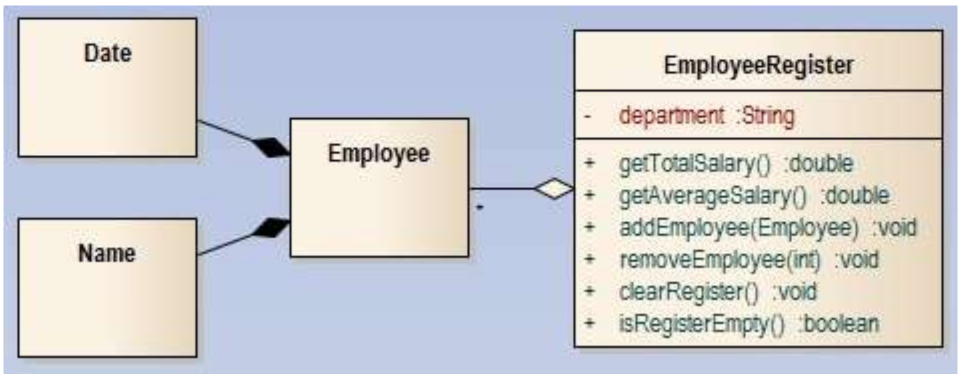
These exercises will help to familiarise you with aggregation relationships where a class primarily acts as a container to hold a collection of other objects. Whilst there are lots of collection classes that represent different data structures, you have so far only been introduced to the ArrayList and so this will be the collection of choice and form one of your fields.

Question 6.4 Download the existing Java Project that can be extracted from the "**Aggregation.zip**" file, and imported into Eclipse. Inside the lib.playlist package, the **PlayList** class aggregates Song objects. The Song class is similar to the CDTrack from week 3.



Study the code and comments in **PlayList**, noting how it holds a collection of Song objects and specifies common methods of an aggregate class, that delegate to the internal data structure, such as add, remove, get, size, etc. There are other methods providing additional functionality, some of which show different implementations (including updated Java 8 stream-based approaches). In the **PlayListApp** program (in the main package), look at how a PlayList is created and used.

Question 6.5 Study the following UML specification and see how you can create a system that combines multiple classes and displays both composition and aggregation relationships. Inside the lib.employeeRegister package, look at the **EmployeeRegister** class, that aggregates Employee objects.


Date
Name
Employee
*
EmployeeRegister
- department :String
+ getTotalSalary() :double
+ getAverageSalary() :double
+ addEmployee(Employee) :void
+ removeEmployee(int) :void
+ clearRegister() :void
+ isRegisterEmpty() :boolean

The demonstration program **EmployeeRegisterDemo** in the main package creates an EmployeeRegister, adds some employees, then prints out all employee details, e.g. a list of employee names, the number of employees and the total and average of all salaries.

Please turn over...

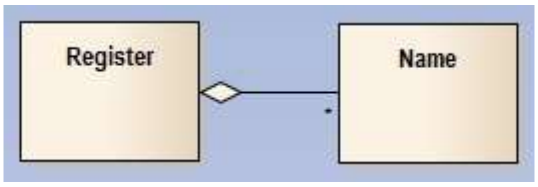
Notes - UML class diagrams:

- Fields that form a composition or aggregation relationship will not be listed explicitly and are instead implied by the relationship itself. Therefore in EmployeeRegister above, there was no need to specify a field: "list : ArrayList<Employee>" because there is an aggregation relationship that already implies that EmployeeRegister holds a collection of type Employee.
- Having viewed composition and aggregation relationships individually, the same principles can be reapplied to create more complex systems consisting of multiple classes and associations

Question 6.6 [✸Portfolio B.1✸]

Download the existing Java Project that can be extracted from the "**PortfolioRegister.zip**" file, and imported into Eclipse. This question forms the first part of Portfolio case study B. Your task is to create a **Register** class that should represent an aggregation of names. You are advised to use the EmployeeRegister class (from the previous question) and the MultipleDice class (shown in Exercise 5) as a guide.

In the test > lib package you have been given a JUnit Test Case **RegisterTest** containing unit tests, which will be used to mark your implementation of the Register class. You can use it to gain progressive feedback on your design, however, some tests will not pass at this stage and are there to test functionality that you will add in next week's exercise. It is important that you use the same method signatures and return types as those specified in the test program, e.g. note the size method is called "registerSize" when invoked in the testRegisterSize unit test.



a) The Register class (in src > lib) should have a single field - an ArrayList of type Name, and a default constructor. Additionally, methods should allow names to be added, removed and retrieved from the register. It should also be possible to retrieve the register's size, clear all entries within it and check if it is empty. A toString() method should be in the standard convention format.

b) Add a method **searchRegisterByFamilyName(...)** that accepts a `String` argument and returns true or false depending on whether a name exists with the respective family name.

c) Add a method **countFirstNameOccurrences(...)** that accepts a `char` argument and returns an `int` signaling the number of occurrences of first names, ending with the provided character.

d) In the src > main package, you will notice a class called **RegisterApp** that has a method called execute, which accepts a Register and a Name object and returns a String. Currently, it simply returns an empty string. It should however do the following:

   - Remove a name at index 1 from the Register `reg`.
   - Add the Name `n` (passed as a parameter to the execute method) to the register.
   - Return a String that contains each name in the register that has a family name of a length >= 5 in the format: "SURNAME, Initial", e.g. "BLOGGS, J", each followed by a new line.

In the test > main package there is a further JUnit Test Case **RegisterAppTest** containing a unit test called testExecute, which assesses whether the execute method behaves correctly.

Please note: Your solution to part d) will be marked by using a different input data set of names within the register to ensure your solution works dynamically based upon any given data set and is not hardcoded in any way.

e) As this is a portfolio question, you should fully document the Register class with Javadoc, i.e. class header, constructor/method description and tags where appropriate.