Association & Aggregation
Vehicle

Passenger

Engine

Bus

Tyre

Car
Vehicle

- Car
- Engine

Passenger

- Bus
- Tyre
Association

• So far we have seen objects being sent messages within a 'main' function.
• However this does not address how objects can communicate with each other.
• In order to do this we need to have links between objects which allow them to communicate
Association

• At this level of class design this is known as an association and these come in three types
  – A one to one association where one object of a class has a link to one other object of a class
  – A one to many association, where one object of a class has links with many objects of a particular class
  – A many to many association, where many objects of one class have links with many objects of a particular class.
• Associations more frequently occur between objects of different classes, but also occur between different objects of the same class
How to draw Associations (UML Notation)

A one-to-one association

A one-to-many association

A many-to-many association

numbers or ranges can be used if known:
a 'zero or 1' to 'exactly 10' associations

a text label with direction indicator
associates with
Direction of message passing

- Associations are generally assumed to be bi-directional. i.e. a message can pass in both directions between objects.
- However in implementation this doesn't have to be the case as shown in the example below
Association in applications

- The previous example doesn't bear much relevance to a real software application.
- The following example shows a more realistic diagram.

The Association in a timetable example
Aggregation v. Inheritance

• A classification hierarchy shows how classes inherit from each other and shows the position in the hierarchy as 'a kind of' relationship.

• i.e. An Estate car is 'a kind of' car and a car is 'a kind of' vehicle.

• Associations also form hierarchies but they are very different from inheritance. These are described by the following terms:
  • Aggregation
  • Composition
  • Part-Whole
  • A Part Of (APO)
  • Has a
  • Containment
Aggregation v. Inheritance (2)

• In this type of hierarchy, classes do not inherit from other classes but are composed of other classes.

• This means that an object of one class may have its representation defined by other objects rather than by attributes.

• The enclosing class does not inherit any attributes or methods from these other included classes.

• This means that this is a relationship (association) between objects.

• An object of the enclosing class is composed wholly or partly of objects of other classes.

• Any object that has this characteristic is known as an aggregation.
**Aggregation v. Containers**

- The commonly used term for this type of relationship is 'containment'.
- However this is semantically different from the idea of a container.
  - In 'containment', a composition hierarchy defines how an object is composed of other objects in a fixed relationship. The aggregate object cannot exist without its components, which will probably be of a fixed and stable number, or at least will vary within a fixed set of possibilities.
  - A 'container' is an object (of a container class) which is able to contain other objects. The existence of the container is independent of whether it actually contains any objects at a particular time, and contained objects will probably be a dynamic and possibly heterogeneous collections (i.e. the objects contained may be of many different classes).
A Containment Example

• For Example A Car
  • will have an engine compartment with an integral component: the engine
  • This is a containment relationship as the engine is an essential part of a car
• Another part of a car is the boot. What is in the boot does not effect the integrity of the car (i.e. what the car is)
  • The boot can contain many different types of objects (tools, shopping etc) but this does not affect the car object.
• Therefore the boot is a container existing independently of its contents.
A common analogy for composition is the exploded parts diagram.

For example, a clock can be thought of as being composed of the following parts:
- Case, Works, Face, Minute Hand, Hour Hand

These objects may exist in many layers, for example, the works is an object made up of many other objects (gears, springs, etc.).
Aggregation or composition?

• An object which comprise parts of a larger object may or may not be visible from outside the object.
• Composition implies that the internal objects are not seen from the outside,
• Whereas aggregation shows that the object may be directly accessed and is visible from outside the object.
• In UML notation this is draw with a Diamond shape and is
  • Filled in to indicate a composition
  • Outlined (left blank) for an aggregation
• This is shown in the following Diagram
UML Diagram of Clock aggregation
Properties of Aggregations

- There are certain properties associated with objects in an aggregation that make them different from normal associations.
- These may be classed as follows

  **Transitivity**
  If A is part of B and B is part of C then A is part of C

  **Antisymmetry**
  If A is part of B, then B is not part of A. (i.e. not a simple association)

  **Propagation**
  The environment of the part is the same as that of the assembly
Properties of Aggregations (2)

• Aggregation can be fixed, variable or recursive
• These may be classed as follows

**Fixed**    The particular numbers and types of the component parts are pre-defined.

**Variable** The number of levels of aggregation is fixed, but the number of parts may vary

**Recursive** The object contains components of its own type. (like a Russian doll)
Aggregation C++ Syntax

• There are two basic ways in which associations and aggregations are implemented

  1. Objects contain Objects
  2. Objects contain pointers to Objects

• The first approach is used to create fixed aggregations (objects inside objects)

• The second is used to create variable aggregations to make programs more flexible
Implementing fixed aggregations

• In C++ fixed aggregations are implemented by defining classes with objects of other classes inside of them.

• For example a simplified aircraft set of components could contain the following

  • PortWing, StarbordWing
  • Engine1, Engine2
  • Fuselage
  • Tailplane

• All of which will be contained as private elements of the class Aircraft as shown in the following class.
• Each of the separate classes within the Aircraft class will have their own methods which can be called within the Aircraft class as shown below
Example of Methods Aircraft Class

void Aircraft::turnToPort()
{
    port_wing.elevatorUp();
    starboard_wing.elevatorUp();
    port_wing.aileronUp();
    starboard_wing.aileronDown();
    tailplane.rudderLeft();
}

• Activities of some composing objects will depend on the state of others
This is an example of a propagation as the state of the Aircraft propagates to the state of the door.

```c++
void Aircraft::openDoors()
{
    if(engine1.getSpeed() > IDLE || engine2.getSpeed() > IDLE)
    {
        // don't open doors
    }
    else
    {
        fuselage.openDoors();
    }
}
```
Constructing Aggregations with parameters

- With aggregation we sometimes have to call parametrised constructors.
- When an object is created, any contained objects must be created at the same time.
- Some objects may not have parametrised constructors so this is easy.
- If some objects do need parameters in the constructors, some mechanism is required to pass the parameter to the associated objects.
- This is done using the colon (:) operator as shown in the following example.
Car Class Comprised of wheel and engine

class Wheel
{
private:
    int diameter;
public:
    Wheel(int diameter_in){ diameter = diameter_in; }
    int getDiameter() { return diameter; }
};
class Engine
{
private:
    int cc;
public:
    Engine(int cc_in) { cc = cc_in; }
    int getCC() return cc; }
}
class Car {
private:
    Wheel nearside_front, offside_front,
    nearside_rear, offside_rear;
    Engine engine;
    int passengers;
public:
    Car(int diameter_in, int cc_in, int passengers_in);
    void showSelf();
};
Car::Car(int diameter_in, int cc_in, int passengers_in) :
    nearside_front(diameter_in),
    offside_front(diameter_in),
    nearside_rear(diameter_in),
    offside_rear(diameter_in),
    engine(cc_in)
{
    passengers = passengers_in;
}