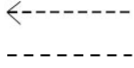
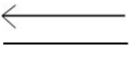
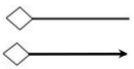

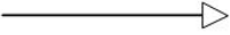



Distinguishing the class diagram association types

Summary of association types:

Type	Direction	Notation	Contextual Examples	Scope	Ownership	Sharing	Lifetime
Dependency / Weak Association	Uni Bi		CustomerView -- Customer	method	No	Yes	Independent of each other
Simple Association	Uni Bi		Student -- Faculty	Object or class	No	Yes	Independent of each other
Aggregation / Weak Composition	Bi Uni		Car – Tyre Team - Player	Object or class	Generally, Yes	Yes	Mostly dependent
Composition	Bi Uni		Polygon -- Side Person -- Heart	Object or class	Always, Yes	No	Strictly dependent

Type	Notation	Context	Example
Generalisation		Concrete super and sub class. If the super class is Abstract, it is represented by italicized name	Rectangle – Square <i>Vehicle</i> – Car
Realisation		Super entity is either an abstract class or an interface	<i>Vehicle</i> -- Car List<T> -- <u>ArrayList<T></u>

Distinction among dependency, association, aggregation and composition is decided by four factors.

- **Scope**
 - the scope defines the part of class body where an instance of an outside class may be accessed.
- **Ownership**
 - an object objA of class A owns an object objB of class B, if objB is "created" inside objA
- **Sharing**
 - an object objB of class B can be accessed/shared by two or more different objects
- **Lifetime**
 - the lifetime of an object objA of class A is defined by its existence in memory, i.e. the moment it is created in computer memory until it is killed.

Rules:

- If the variable **scope** is limited to a few methods (method-level **scope**), then the association is *weak association* or *dependency*.
- If the **scope** is object-level (see example below), the relation at the least is *simple association* (aka *normal association* or simply *association*).
 - If the **scope** is object-level, but the class does not **own** this object, then the relation is generally *simple association*. In the Example 2, the class Engine's object is passed to the constructor of class Car from outside this class and hence the class Car's object does not own this object.
 - If a class' object **owns** (object of latter is created inside the former) another class' object as an object-level variable, then we have at least an *aggregation* relation. In Example 3, each object of class Car will own an object of class Engine, as the object of class Engine is created inside class Car.
 - If the **owner** class **shares** the reference of the **owned** class' object with other classes' objects, then we can not have *composition* and hence this is most likely a case of *aggregation*. We need to still make sure that the **lifetimes** of the two also match. Note for the **lifetime** however, the matching is not strict i.e. destroying a *part* object (e.g. an object of class Engine in the examples above) may not destroy the *whole* object (e.g. an object of class Car in the examples above) and vice-versa.
 - If one class **owns** another class, and also does not **share** the reference of its objects with other classes, and also has a strictly dependent **lifetime** (destroying at least one destroys the other), then we have a **composition** relationship. In Example 4, each object of class Car not only owns an object of class Engine, but also does not share its reference with any other objects (e.g. objects of other classes). If we destroy the object of class Car, then the Engine object will also become useless, as no other objects reference it.

Examples:

```
class Car
{
    Car(Engine engine)
    {
        engine.start();
    }

    public void move()
    {
        //some code with no reference to Engine
    }
}

class Engine
{
    public void start()
    {
        System.out.println("Start the engine...");
    }
}
```

Example 1: Car has Weak Association with or Dependency over Engine

```
class Car
{
    private Engine _engine;

    Car(Engine engine)
    {
        _engine = engine;
    }

    public void move()
    {
        if(_engine != null)
        {
            _engine.start();
            System.out.println("Car moves... ");
        }
    }
}

class Engine
{
    public void start()
    {
        System.out.println("Start the engine...");
    }
}
```

Example 2: Car has Simple Association with Engine

```

class Car
{
    private Engine _engine;

    Car()
    {
        _engine = new Engine();
    }

    public void move()
    {
        if(_engine != null)
        {
            Driver driver = new Driver(_engine);
            //shares a reference of Engine object to an external object
            _engine.start();
            System.out.println("Car moves... ");
        }
    }
}

class Engine
{
    public void start()
    {
        System.out.println("Start the engine...");
    }
}

```

Example 3: Car has Aggregation with Engine

```

class Car
{
    private Engine _engine;

    Car()
    {
        _engine = new Engine();
    }

    public void move()
    {
        if(_engine != null)
        {
            _engine.start();
            System.out.println("Car moves... ");
        }
    }
}

class Engine
{
    public void start()
    {
        System.out.println("Start the engine...");
    }
}

```

Example 4: Car has Composition with Engine

For more details about distinguishing various types of class associations, refer Lecture 19-20 study material (especially the slides describing different association types and those listing different distinguishing factors of scope, ownership, lifetime and sharing), and Week 7 practice exercises.

Happy Coding,

~Paramvir Singh

(p.singh@auckland.ac.nz)