Aggregation and Composition

Both aggregation and composition represent a whole-part (has-a/part-of) association.

The main differentiator between aggregation and composition is the lifecycle dependence between whole and part.

In aggregation, the part may have an independent lifecycle, it can exist independently. When the whole is destroyed the part may continue to exist.

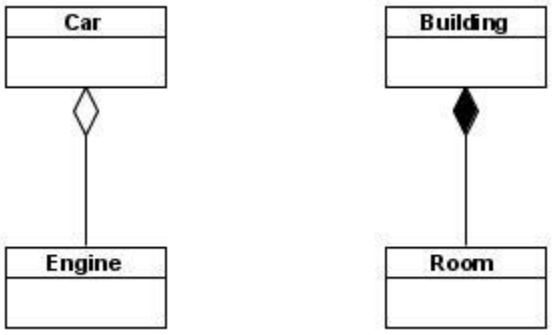
For example, a car has many parts. A part can be removed from one car and installed into a different car. If we consider a salvage business, before a car is destroyed, they remove all saleable parts. Those parts will continue to exist after the car is destroyed.

Composition is a stronger form of aggregation. The lifecycle of the part is strongly dependent on the lifecycle of the whole. When the whole is destroyed, the part is destroyed too.

For example, a building has rooms. A room can exist only as part of a building. The room cannot be

removed from one building and attached to a different one. When the building ceases to exist so do all rooms that are part of it.

Class Diagram



Design

When we decide between aggregation and composition we need to answer the following question: can the part exist independently? Also we need to make sure we do fulfill the business requirements (which sometimes may enforce a whole-part association that may seem counterintuitive). For example, the car-part relationship mentioned above is an aggregation, but if the business requirements state that "no part of a vehicle can be used for resale or as spare part; at the end of the amortization period the vehicle is

destroyed" this will suggest a composition relationship between car and its parts.

Implementation in Java

Both the whole and the part are Java classes which at runtime will be represented by Java objects. We decide the nature of whole-part relationship by how the part objects are created and destroyed.

In Java, objects do not need to be destroyed explicitly; an object is automatically "destroyed" when it is garbage collected. An object is garbage collected when it is not referenced by any other object. Regardless of the type of relationship - aggregation or composition - the part is referenced by the whole. If the whole is destroyed (garbage collected), then the part is no longer referenced by the whole. If there is no other object referencing the part then the part is destroyed too.

To summarize, in Java the difference between aggregation and composition boils down to the following: is the variable that holds the part object accessible to objects other than the whole? If the

answer is yes, then we have aggregation, if the answer is no then we have composition.

In order to enforce a composition relationship we need to make sure that the part object is accessible only by the whole object, which means that:

- The part object must be created inside the whole object (either inside constructor, as part of a static block or some init method which is invoked when the whole is created.)
- The instance variable that holds the part must be private
- There is no getter/setter for accessing the part (there is no way an outside object has access to the part object)

The following two examples show the Java implementation of aggregation and composition. We used the same Car - Engine relationship for both aggregation and composition in order to illustrate the programming changes required in order to convert an aggregation relationship into composition.

Aggregation - Java sample code

```
public class Car {
	private String make;
	private int year;
	private Engine engine;

	public Car(String make, int year, Engine engine)
{
		this.make = make;
		this.year = year;
		// the engine object is created outside and
is passed as argument to Car constructor
		// When this Car object is destroyed, the
engine is still available to objects other than Car
		// If the instance of Car is garbage
collected the associated instance of Engine may not
be garbage collected (if it is still referenced by other
objects)
		this.engine = engine;
	}
```

```
	public String getMake() {
		return make;
	}

	public int getYear() {
		return year;
	}

	public Engine getEngine() {
		return engine;
	}
}

public class Engine {
	private int engineCapacity;
	private int engineSerialNumber;

	public Engine(int engineCapacity, int
engineSerialNumber) {
		this.engineCapacity = engineCapacity;
		this.engineSerialNumber =
engineSerialNumber;
```

```
	}

	public int getEngineCapacity() {
		return engineCapacity;
	}

	public int getEngineSerialNumber() {
		return engineSerialNumber;
	}
}
```

Composition - Java sample code

```
public class Car {
	private String make;
	private int year;
	private Engine engine;

	public Car(String make, int year, int
engineCapacity, int engineSerialNumber) {
		this·make=make;
```

```
        this.year=year;
        // we create the engine using parameters
passed in Car constructor
        // only the Car instance has access to the
engine instance
        // when Car instance is garbage collected,
the engine instance is garbage collected too
        engine = new Engine(engineCapacity,
engineSerialNumber);
    }

    public String getMake() {
        return make;
    }

    public int getYear() {
        return year;
    }

    public int getEngineSerialNumber() {
        return engine.getEngineSerialNumber();
    }
```

```
	public int getEngineCapacity() {
		return engine.getEngineCapacity();
	}
}
```