10.3 — Aggregation

BY ALEX ON DECEMBER 7TH, 2007 | LAST MODIFIED BY ALEX ON DECEMBER 7TH, 2007

In the previous lesson on **composition**, you learned that compositions are complex classes that contain other subclasses as member variables. In addition, in a composition, the complex object "owns" all of the subobjects it is composed of. When a composition is destroyed, all of the subobjects are destroyed as well. For example, if you destroy a car, it's frame, engine, and other parts should be destroyed as well. If you destroy a PC, you would expect it's RAM and CPU to be destroyed as well.

Aggregration

An **aggregation** is a specific type of composition where no ownership between the complex object and the subobjects is implied. When an aggregate is destroyed, the subobjects are not destroyed.

For example, consider the math department of a school, which is made up of one or more teachers. Because the department does not own the teachers (they merely work there), the department should be an aggregate. When the department is destroyed, the teachers should still exist independently (they can go get jobs in other departments).

Because aggregations are just a special type of compositions, they are implemented almost identically, and the difference between them is mostly semantic. In a composition, we typically add our subclasses to the composition using either normal variables or pointers where the allocation and deallocation process is handled by the composition class.

In an aggregation, we also add other subclasses to our complex aggregate class as member variables. However, these member variables are typically either references or pointers that are used to point at objects that have been created outside the scope of the class. Consequently, an aggregate class usually either takes the objects it is going to point to as constructor parameters, or it begins empty and the subobjects are added later via access functions or operators.

Because these subclass objects live outside of the scope of the class, when the class is destroyed, the pointer or reference member variable will be destroyed, but the subclass objects themselves will still exist.

Let's take a look at our Teacher and Department example in more detail.

```
#include <string>
using namespace std;

class Teacher
{
private:
    string m_strName;
public:
    Teacher(string strName)
        : m_strName(strName)
    {
    }

    string GetName() { return m_strName; }
};

class Department
{
private:
    Teacher *m_pcTeacher; // This dept holds only one teacher

public:
    Department(Teacher *pcTeacher=NULL)
        : m_pcTeacher(pcTeacher)
    {
    }
};

```

```
int main()
{
    // Create a teacher outside the scope of the Department
    Teacher *pTeacher = new Teacher("Bob"); // create a teacher
    {
        // Create a department and use the constructor parameter to pass
        // the teacher to it.
        Department cDept(pTeacher);

    } // cDept goes out of scope here and is destroyed

    // pTeacher still exists here because cDept did not destroy it
    delete pTeacher;
}
```

In this case, pTeacher is created independetly of cDept, and then passed into cDept's constructor. Note that the department class uses an initialization list to set the value of m_pcTeacher to the pTeacher value we passed in. When cDept is destroyed, the m_pcTeacher pointer destroyed, but pTeacher is not deallocated, so it still exists until it is independently destroyed.

To summarize the differences between composition and aggregation:

Compositions:

- Typically use normal member variables
- Can use pointer values if the composition class automatically handles allocation/deallocation
- Responsible for creation/destruction of subclasses

Aggregations:

- Typically use pointer variables that point to an object that lives outside the scope of the aggregate class
- Can use reference values that point to an object that lives outside the scope of the aggregate class
- Not responsible for creating/destroying subclasses

It is worth noting that the concepts of composition and aggregation are not mutually exclusive, and can be mixed freely within the same class. It is entirely possible to write a class that is responsible for the creation/destruction of some subclasses but not others. For example, our Department class could have a name and a teacher. The name would probably be added to the department by composition, and would be created and destroyed with the department. On the other hand, the teacher would be added to the department by aggregate, and created/destroyed independently.

It is also possible to create other hybrid aggregate/composition schemes, such as where a class holds independent subobjects like an aggregate, but will destroy them when the class goes out of scope like a composition.

While aggregates can be extremely useful (which we will see more of in the next lesson on container classes), they are also potentially dangerous. As noted several times, aggregates are not responsible for deallocating their subobjects when they are destroyed. Consequently, if there are no other pointers or references to those subobjects when the aggregate is destroyed, those subobjects will cause a memory leak. It is up to the programmer to ensure that this does not happen. This is generally handled by ensuring other pointers or references to those subobjects exist when the aggregate is destroyed.

 **10.4 -- Container classes**

 **Index**

 **10.2 -- Composition**