10.4 — Association

BY ALEX ON AUGUST 19TH, 2016 | LAST MODIFIED BY ALEX ON SEPTEMBER 25TH, 2017

In the previous two lessons, we've looked at two types of object composition, composition and aggregation. Object composition is used to model relationships where a complex object is built from one or more simpler objects (parts).

In this lesson, we'll take a look at a weaker type of relationship between two otherwise unrelated objects, called an association. Unlike object composition relationships, in an association, there is no implied whole/part relationship.

Association

To qualify as an **association**, an object and another object must have the following relationship:

- The associated object (member) is otherwise unrelated to the object (class)
- The associated object (member) can belong to more than one object (class) at a time
- The associated object (member) does *not* have its existence managed by the object (class)
- The associated object (member) may or may not know about the existence of the object (class)

Unlike a composition or aggregation, where the part is a part of the whole object, in an association, the associated object is otherwise unrelated to the object. Just like an aggregation, the associated object can belong to multiple objects simultaneously, and isn't managed by those objects. However, unlike an aggregation, where the relationship is always unidirectional, in an association, the relationship may be unidirectional or bidirectional (where the two objects are aware of each other).

The relationship between doctors and patients is a great example of an association. The doctor clearly has a relationship with his patients, but conceptually it's not a part/whole (object composition) relationship. A doctor can see many patients in a day, and a patient can see many doctors (perhaps they want a second opinion, or they are visiting different types of doctors). Neither of the object's lifespans are tied to the other.

We can say that association models as "uses-a" relationship. The doctor "uses" the patient (to earn income). The patient uses the doctor (for whatever health purposes they need).

Implementing associations

Because associations are a broad type of relationship, they can be implemented in many different ways. However, most often, associations are implemented using pointers, where the object points at the associated object.

In this example, we'll implement a bi-directional Doctor/Patient relationship, since it makes sense for the Doctors to know who their Patients are, and vice-versa.

```cpp
#include <iostream>
#include <string>
#include <vector>

// Since these classes have a circular dependency, we're going to forward declare Doctor
class Doctor;

class Patient
{
private:
    std::string m_name;
    std::vector<Doctor *> m_doctor; // so that we can use it here

    // We're going to make addDoctor private because we don't want the public to use it.
    // They should use Doctor::addPatient() instead, which is publicly exposed
    // We'll define this function after we define what a Doctor is
    // Since we need Doctor to be defined in order to actually use anything from it
    void addDoctor(Doctor *doc);

public:
    Patient(std::string name)
        : m_name(name)
    {
```

```
    }

    // We'll implement this function below Doctor since we need Doctor to be defined at that point
    friend std::ostream& operator<<(std::ostream &out, const Patient &pat);

    std::string getName() const { return m_name; }

    // We're friending Doctor so that class can access the private addDoctor() function
    // (Note: in normal circumstances, we'd just friend that one function, but we can't
    // because Doctor is forward declared)
    friend class Doctor;
};

class Doctor
{
private:
    std::string m_name;
    std::vector<Patient *> m_patient;

public:
    Doctor(std::string name):
        m_name(name)
    {
    }

    void addPatient(Patient *pat)
    {
        // Our doctor will add this patient
        m_patient.push_back(pat);

        // and the patient will also add this doctor
        pat->addDoctor(this);
    }


    friend std::ostream& operator<<(std::ostream &out, const Doctor &doc)
    {
        unsigned int length = doc.m_patient.size();
        if (length == 0)
        {
            out << doc.m_name << " has no patients right now";
            return out;
        }

        out << doc.m_name << " is seeing patients: ";
        for (unsigned int count = 0; count < length; ++count)
             out << doc.m_patient[count]->getName() << ' ';

        return out;
    }

    std::string getName() const { return m_name; }
};

void Patient::addDoctor(Doctor *doc)
{
    m_doctor.push_back(doc);
}

std::ostream& operator<<(std::ostream &out, const Patient &pat)
{
    unsigned int length = pat.m_doctor.size();
    if (length == 0)
    {
        out << pat.getName() << " has no doctors right now";
        return out;
    }
```

```cpp

    out << pat.m_name << " is seeing doctors: ";
    for (unsigned int count = 0; count < length; ++count)
        out << pat.m_doctor[count]->getName() << ' ';

    return out;
}


int main()
{
    // Create a Patient outside the scope of the Doctor
    Patient *p1 = new Patient("Dave");
    Patient *p2 = new Patient("Frank");
    Patient *p3 = new Patient("Betsy");

    Doctor *d1 = new Doctor("James");
    Doctor *d2 = new Doctor("Scott");

    d1->addPatient(p1);

    d2->addPatient(p1);
    d2->addPatient(p3);

    std::cout << *d1 << '\n';
    std::cout << *d2 << '\n';
    std::cout << *p1 << '\n';
    std::cout << *p2 << '\n';
    std::cout << *p3 << '\n';

    delete p1;
    delete p2;
    delete p3;

    delete d1;
    delete d2;

    return 0;
}
```

This prints:

```
James is seeing patients: Dave
Scott is seeing patients: Dave Betsy
Dave is seeing doctors: James Scott
Frank has no doctors right now
Betsy is seeing doctors: Scott
```

In general, you should avoid bidirectional associations if a unidirectional one will do, as they add complexity and tend to be harder to write without making errors.

Reflexive association

Sometimes objects may have a relationship with other objects of the same type. This is called a **reflexive association**. A good example of a reflexive association is the relationship between a university course and its prerequisites (which are also university courses).

Consider the simplified case where a Course can only have one prerequisite. We can do something like this:

```cpp
#include <string>
class Course
{
private:
    std::string m_name;
    Course *m_prerequisite;
```

```

public:
    Course(std::string &name, Course *prerequisite=nullptr):
        m_name(name), m_prerequisite(prerequisite)
    {
    }

};
```

This can lead to a chain of associations (a course has a prerequisite, which has a prerequisite, etc…)

Associations can be indirect

In all of the above cases, we’ve used a pointer to directly link objects together. However, in an association, this is not strictly required. Any kind of data that allows you to link two objects together suffices. In the following example, we show how a Driver class can have a unidirectional association with a Car without actually including a Car pointer member:

```
#include <iostream>
#include <string>

class Car
{
private:
    std::string m_name;
    int m_id;

public:
    Car(std::string name, int id)
        : m_name(name), m_id(id)
    {
    }

    std::string getName() { return m_name; }
    int getId() { return m_id;  }
};

// Our CarLot is essentially just a static array of Cars and a lookup function to retrieve them.
// Because it's static, we don't need to allocate an object of type CarLot to use it
class CarLot
{
private:
    static Car s_carLot[4];

public:
    CarLot() = delete; // Ensure we don't try to allocate a CarLot

    static Car* getCar(int id)
    {
        for (int count = 0; count < 4; ++count)
            if (s_carLot[count].getId() == id)
                return &(s_carLot[count]);

        return nullptr;
    }
};

Car CarLot::s_carLot[4] = { Car("Prius", 4), Car("Corolla", 17), Car("Accord", 84), Car("Matrix", 62) }
;

class Driver
{
private:
    std::string m_name;
    int m_carId; // we're associated with the Car by ID rather than pointer

public:
    Driver(std::string name, int carId)
```

```
            : m_name(name), m_carId(carId)
        {
        }

        std::string getName() { return m_name; }
        int getCarId() { return m_carId; }

};

int main()
{
    Driver d("Franz", 17); // Franz is driving the car with ID 17

    Car *car = CarLot::getCar(d.getCarId()); // Get that car from the car lot

    if (car)
        std::cout << d.getName() << " is driving a " << car->getName() << '\n';
    else
        std::cout << d.getName() << " couldn't find his car\n";

    return 0;
}
```

In the above example, we have a CarLot holding our cars. The Driver, who needs a car, doesn’t have a pointer to his Car -- instead, he has the ID of the car, which we can use to get the Car from the CarLot when we need it.

In this particular example, doing things this way is kind of silly, since getting the Car out of the CarLot requires an inefficient lookup (a pointer connecting the two is much faster). However, there are advantages to referencing things by a unique ID instead of a pointer. For example, you can reference things that are not currently in memory (maybe they’re in a file, or in a database, and can be loaded on demand). Also, pointers can take 4 or 8 bytes -- if space is at a premium and the number of unique objects is fairly low, referencing them by an 8-bit or 16-bit integer can save lots of memory.

Composition vs aggregation vs association summary

Here’s a summary table to help you remember the difference between composition, aggregation, and association:

Property	Composition	Aggregation	Association
Relationship type	Whole/part	Whole/part	Otherwise unrelated
Members can belong to multiple classes	No	Yes	Yes
Members existence managed by class	Yes	No	No
Directionality	Unidirectional	Unidirectional	Unidirectional or bidirectional
Relationship verb	Part-of	Has-a	Uses-a

10.5 -- Dependencies

Index

10.3 -- Aggregation