Part C - Class Relationships

Compositions, Aggregations and Associations

Design collections of model objects
Distinguish class relationships with respect to ownership

"Prefer composition to inheritance" Sutter, Alexandruscu (2005).

Compositions | Aggregations | Associations | Exercises

The relationships between classes in object-oriented applications that specify ownership extent include compositions, aggregations and associations. Each of these relationships reflects a different degree of coupling between the classes. A composition is a strong relationship: the composer object owns the component object: one class completely contains another class and determines its lifetime. An aggregation is a weaker relationship: the aggregator has an instance of another class, which determines its own lifetime. An association is the weakest relationship of the three: one class accesses or uses another class: neither class has the other class.

If a class with a resource is responsible for copying and destroying its resource, then that class is a composition. If the class is not responsible for copying or destroying its resource, then that class is an aggregation or an association.

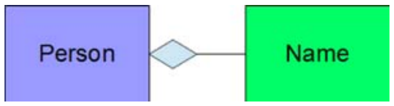
This chapter presents examples of composition, aggregation and association relationships between classes.

COMPOSITIONS

A composition is a *has-a* relationship between classes that implements complete ownership. The composer object is responsible for destroying its component object(s) at or before the time of its own destruction. A composition is incomplete without its components.

Composition is more flexible (less coupled) design-wise than inheritance. Updates to the component class need not affect the composer class. However, member functions added to the component class require forwarding member functions in the composer class.

Consider the composition relationship between a **Person** class and a **Name** class illustrated below. Every person has a name and an age.



The class definition and implementation for the **Name** type are listed on the left and right respectively:

```
#ifndef NAME_H
#define NAME_H
// Composition - Name
// Name.h

class Name {
    char* name = nullptr;
```

```
// Composition - Name
// Name.cpp

#include <cstring>
#include "Name.h"

Name::Name(const char* n)  :
  public:
    Name(const char*);
    Name(const Name&);
    Name& operator=(const Name&);
    ~Name();
    const char* get() const;
    void set(const char*);
};
#endif
```

```
 name {new char[std::strlen(n) + 1]} {
    std::strcpy(name, n);
}
Name::Name(const Name& src) {
    *this = src;
}
Name& Name::operator=(const Name&  src) {
    if (this != &src) {
        delete [] name;
        name = new
         char[std::strlen(src.name) + 1];
        std::strcpy(name, src.name);
    }
    return *this;
}
Name::~Name() { delete [] name; }
const char* Name::get() const {
    return name;
}
void Name::set(const char* n) {
    delete [] name;
    name = new char[std::strlen(n) + 1];
    std::strcpy(name, n);
}
```

We implement this composition using either a **Name** subobject or a pointer to a **Name** object. The subobject version is listed on the left; the pointer version is listed on the right:

```
// Composition - SubObject Version
// Person.h

#include "Name.h"

class Person {
    Name name; // subobject
    int age;
  public:
    Person(const char*, int);



    void display() const;
    void set(const char*);
    //...
};
```

```
// Composition - Pointer Version
// Person.h

class Name;

class Person {
    Name* name = nullptr; // pointer
    int age;
  public:
    Person(const char*, int);
    Person(const Person&);
    Person& operator=(const Person&);
    ~Person();
    void display() const;
    void set(const char*);
    //...
};
```

The implementation files for both versions are listed below. The **Name** object does not exist apart from the **Person** object. The **Person** constructor creates the **Name** object, the assignment operator destroys the old object and creates a new one, and the destructor destroys the object. In the subobject version, the default copying and assignment rules apply: the default copy constructor, assignment operator and destructor are sufficient. In the pointer version, deep copying and assignment are required and we must code the copy constructor, assignment operator and destructor.

```
// Composition - SubObject Version
// Person.cpp

#include <iostream>
#include "Person.h"
```

```
// Composition - Pointer Version
// Person.cpp

#include <iostream>
#include "Person.h"
#include "Name.h"
```

```
Person::Person(const char* n, int a) :
 name{n}, age{a} {}
void Person::display() const {
    std::cout << age << ' ' <<
     name.get() << std::endl;
}
void Person::set(const char* n) {
   name.set(n);
}
//...
```

```
Person::Person(const char* n, int a) :
 name {new Name(n)}, age {a} {}
Person::Person(const Person& src) {
    *this = src;
}
Person& Person::operator=(const Person&
 src) {
    if (this != &src) {
        delete name;
        name = new Name(*src.name);
        age = src.age;
    }
    return *this;
}
Person::~Person() { delete name; }
void Person::display() const {
    std::cout << age << ' ' <<
     name->get() << std::endl;
}
void Person::set(const char* n) {
    name->set(n);
}
//...
```

The following program produces the output listed on the right for both versions of the **Person** type:

```
// Composition
// composition.cpp

#include "Person.h"

int main() {
    Person p("Harvey", 23);
    Person q = p;
    p.display();                              23 Harvey
    q.display();                              23 Harvey
    q.set("Lawrence");
    p.display();                              23 Harvey
    q.display();                              23 Lawrence
    p = q;
    p.display();                              23 Lawrence
}
```

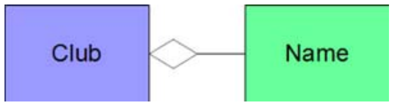
Note that this program is unaware of the **Name** type. The program makes no reference to the types of subobjects contained in the **Person** type. Changes to these objects and the descriptions of their type(s) are completely hidden within the **Person** type.

AGGREGATIONS

An aggregation is a composition that does not manage the creation or destruction of the objects that it *uses*. The responsibility for creating and destroying the objects lies outside the aggregator type. The aggregator is complete whether or not any of the objects that it uses exist. The objects survive the destruction of the aggregator.

Aggregation is more flexible (less coupled) design-wise than composition. Updates to any aggregatee type do not interfere with the design of the aggregator type. Member functions added to the aggregatee type do not require forwarding member functions in the aggregator type.

Consider the relationship between a club and the names of its members as illustrated below. The club has or may have members, but can exist without any. A name can be removed from its list of members before the club is disbanded and that name is not destroyed if the club is disbanded.



The class definition and implementation for a `Club` type might look like:

```
// Aggregation
// Club.h

class Name;
const int M = 50;

class Club {
    const Name* name[M]{};
    int m = 0;
  public:
    Club& operator+=(const Name&);
    Club& operator-=(const Name&);
    void display() const;
    //...
};
```

```
// Aggregation
// Club.cpp

#include <iostream>
#include <cstring>
#include "Club.h"
#include "Name.h"

Club& Club::operator+=(const Name& n) {
    if (m < M)
        name[m++] = &n;
    return *this;
}
Club& Club::operator-=(const Name& t) {
    bool found = false;
    int i;
    for (i = 0; i < m && !found; i++)
        if (!std::strcmp(name[i]->get(),
         t.get())) found = true;
    if (found) {
        for (; i < m; i++)
            name[i - 1] = name[i];
        if (m) {
            name[m - 1] = nullptr;
            m--;
        }
    }
    return *this;
}
void Club::display() const {
    for (int i = 0; i < m; i++)
        std::cout << name[i]->get()
         << std::endl;
}
//...
```

The following program adds the names of four members to a club, removes two names and generates the output listed on the right:

```
// Aggregation
// aggregation.cpp

#include "Club.h"
#include "Name.h"

int main() {
    Name jane("Jane");
    Name john("John");
```

```
        Name alice("Alice");
        Name frank("Frank");
        Name stanley("Stanley");
        Club c;
        c += jane;
        c += john;
        c += alice;
        c += frank;                                           Jane
        c.display();                                          John
        c -= alice;                                           Alice
        c -= john;                                            Frank
        c -= stanley;
        c.display();                                          Jane
    }                                                         Frank
```

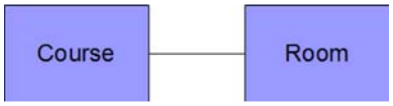
The application creates the **`Name`** objects separately from the **`Club`** and destroys them separately.

ASSOCIATIONS

An association is a service relationship and does not involve ownership of one type by another. Each type is wholly independent and complete without the related type.

Association is the least coupled relationship between classes. Member functions added to an associating type do not require forwarding member functions in the related type.

Consider the relationship between a course and a room. A course uses a room and a room is booked for a course, but both exist independently of one another. A room can be booked for a course and a course can be assigned to a room. Neither is destroyed when the other is destroyed.



The class definition and implementation for a **`Course`** type might look like:

```
// Association
// Course.h

#include "Name.h"
class Room;

class Course {
    Name name;
    int code;
    Room* room = nullptr;
  public:
    Course(const char*, int);
    void book(Room&);
    void unbook();
    const char* get() const;
    void display() const;
    //...
};
```

```
// Association
// Course.cpp

#include <iostream>
#include "Course.h"
#include "Room.h"

Course::Course(const char* n, int c) :
 name{n}, code{c} {}
void Course::book(Room& r) {
    if (room) room->unbook();
    room = &r;
}
void Course::unbook() {
    room = nullptr;
}
const char* Course::get() const {
    return name.get();
}
void Course::display() const {
    std::cout <<
     (room ? room->get() : "*****")
     << ' ' << code << ' ' << name.get()
    << std::endl;
}
//...
```

The class definition and implementation for a **`Room`** type might look like:

```
// Association
// Room.h

#include "Name.h"
class Course;

class Room {
    Name name;
    Course* course = nullptr;
  public:
    Room(const char*);
    void book(Course&);
    void unbook();
    const char* get() const;
    void display() const;
    //...
};
```

```
// Association
// Room.cpp

#include <iostream>
#include "Room.h"
#include "Course.h"

Room::Room(const char* n) : name{n} {}
void Room::book(Course& c) {
    if (course) course->unbook();
    course = &c;
}
void Room::unbook() {
    course = nullptr;
}
const char* Room::get() const {
    return name.get();
}
void Room::display() const {
    std::cout << name.get() << ' ' <<
     (course ? course->get() : "available")
     << std::endl;
}
//...
```

The following program assigns two of three courses to two of three rooms leaving one course unassigned and one room unbooked:

```
// Association
// association.cpp

#include "Course.h"
#include "Room.h"

void book(Course& c, Room& r) {
    c.book(r);
    r.book(c);
}

int main() {
    Room t2108("T2108");
    Room t2109("T2109");
    Room t2110("T2110");
    Course btp105("Intro to Programming", 105);
    Course btp205("Intro to O-O Prg", 205);
    Course btp305("O-O Programming", 305);
    btp105.display();                              ***** 105 Intro to Programming
    btp205.display();                              ***** 205 Intro to O-O Prg
    btp305.display();                              ***** 305 O-O Programming
    t2108.display();                               T2108 available
    t2109.display();                               T2109 available
    t2110.display();                               T2110 available
    book(btp205, t2110);
    book(btp305, t2108);
    btp105.display();                               ***** 105 Intro to Programming
    btp205.display();                               T2110 205 Intro to O-O Prg
    btp305.display();                               T2108 305 O-O Programming
    t2108.display();                                T2108 O-O Programming
    t2109.display();                                T2109 available
    t2110.display();                                T2110 Intro to O-O Programming
    book(btp205, t2108);
    book(btp305, t2109);
    btp105.display();                               ***** 105 Intro to Programming
    btp205.display();                               T2108 205 Intro to O-O Prg
    btp305.display();                               T2109 305 O-O Programming
    t2108.display();                                T2108 Intro to O-O Programming
    t2109.display();                                T2109 O-O Programming
    t2110.display();                                T2110 available
}
```

EXERCISES

- Read StackOverFlow on Difference Between Aggregation and Composition
- Read Software Engineering Stack Exchange on Differences between Association, Aggregation and Composition