

# Software Development

ICT Home Outline Timeline Notes IPC144 OOP244 Workshops Assignments Instructor

OOP345

## 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, Alexandrescu (2005).*

[Compositions](#) | [Aggregations](#) | [Associations](#) | [Exercises](#)

The relationships between classes in object-oriented applications, aside from inheritance and parametric polymorphism, exhibit different degrees of ownership. These relationships include compositions, aggregations and associations. Each relationship reflects a different degree of coupling between 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 exhibit a 'has a' relationship to the other class.

These relationships appear in all three forms in the case of class with resources. 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.

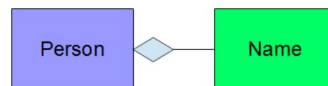
This chapter presents examples of each of these three relationships.

## COMPOSITIONS

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

Design-wise, composition is more flexible (less coupled) 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 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 };
public:
    Name(const char*);
    Name(const Name&);
    Name& operator=(const Name&);
    ~Name();
    const char* get() const;
    void set(const char*);
};
#endif

// Composition - Name
// Name.cpp

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

Name::Name(const char* n) :
    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
  
```

Welcome

Notes

Overview

Building Blocks

Compilation

Fundamental Types

Built-In Types

Class Types

Inheritance

Templates

Compositions

Expressions

Functions

Error Handling

Standard Library

Containers, Iterators

Algorithms

File Streams

Raw Pointers

Smart Pointers

Multithreading

Thread Classes

Pre-Processor

Arrays Pointers

Multiple Inheritance

Bit Wise Expressions

Linked Lists

Other Topics

Appendices

Workshops

Assignments

Handouts

Practice

Resources

```

        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:

<pre> // 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*);     //... }; </pre>	<pre> // Composition - Pointer Version // Person.h  class Name;  class Person {     Name* name { nullptr }; // pointer     int age; public:     Person(const char*, int);     Person(const Person&amp;);     Person&amp; operator=(const Person&amp;);     ~Person();     void display() const;     void set(const char*);     //... }; </pre>
--	--

The implementation files for both versions are listed below. The **Name** object does not exist apart from the **Person** 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. The **Person** constructor creates the **Name** object, the assignment operator destroys the old **Name** object and creates a new one, and the destructor destroys the **Name** object.

<pre> // Composition - SubObject Version // Person.cpp  #include &lt;iostream&gt; #include "Person.h"  Person::Person(const char* n, int a) :     name{n}, age{a} {}  void Person::display() const {     std::cout &lt;&lt; age &lt;&lt; ' ' &lt;&lt;         name.get() &lt;&lt; std::endl; } void Person::set(const char* n) {     name.set(n); // forwarding } //... </pre>	<pre> // Composition - Pointer Version // Person.cpp  #include &lt;iostream&gt; #include "Person.h" #include "Name.h"  Person::Person(const char* n, int a) :     name {new Name(n)}, age {a} {} Person::Person(const Person&amp; src) {     *this = src; } Person&amp; Person::operator=(const Person&amp; src) {     if (this != &amp;src) {         delete name;         name = new Name(*src.name);         age = src.age;     }     return *this; } Person::~Person() { delete name; } void Person::display() const {     std::cout &lt;&lt; age &lt;&lt; ' ' &lt;&lt;         name-&gt;get() &lt;&lt; std::endl; } void Person::set(const char* n) {     name-&gt;set(n); // forwarding } //... </pre>
--	--

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
}
```

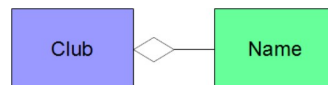
Note that this program is unaware of the implementation of the composition relationship. It makes no reference to the types of subobjects contained in the **Person** type. Changes to these objects and the descriptions of their type(s) are completely hidden within the **Person** type.

## 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 used survive the destruction of the aggregator.

Design-wise, aggregation is more flexible (less coupled) 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 its members. The relationship is between the club and the names of its members as illustrated below. The club has or may have members, but can exist without any. A member's 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;
constexpr 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 gameClub;
    gameClub += jane;
    gameClub += john;
    gameClub += alice;
    gameClub += frank;
    gameClub.display();
    gameClub -= alice;
    gameClub -= john;
    gameClub -= stanley;
    gameClub.display();
}

```

Jane  
 John  
 Alice  
 Frank  
  
 Jane  
 Frank

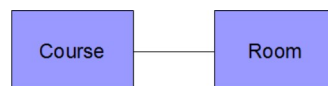
Note how the application creates the **Name** objects separately from the **Club** and destroys them separately.

## ASSOCIATIONS

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

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

Consider the relationship between a course and a room in a college. The course uses the room and the room is booked for the course for a certain period, 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:

<pre> // 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&amp;);     void release();     const char* get() const;     void display() const; </pre>	<pre> // Association // Course.cpp  #include &lt;iostream&gt; #include "Course.h" #include "Room.h"  Course::Course(const char* n, int c) :     name{n}, code{c} {} void Course::book(Room&amp; r) {     if (room) room-&gt;release();     room = &amp;r; } void Course::release() {     room = nullptr; } </pre>
--	---

```

//...
};

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:

<pre> // Association // Room.h  #include "Name.h" class Course;  class Room {     Name name;     Course* course { nullptr }; public:     Room(const char*);     void book(Course&amp;);     void release();     const char* get() const;     void display() const;     //... }; </pre>	<pre> // Association // Room.cpp  #include &lt;iostream&gt; #include "Room.h" #include "Course.h"  Room::Room(const char* n) : name{n} {} void Room::book(Course&amp; c) {     if (course) course-&gt;release();     course = &amp;c; }  void Room::release() {     course = nullptr; }  const char* Room::get() const {     return name.get(); }  void Room::display() const {     std::cout &lt;&lt; name.get() &lt;&lt; ' ' &lt;&lt;         (course ? course-&gt;get() : "available")         &lt;&lt; std::endl; } //... </pre>
--	--

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();
    btp205.display();
    btp305.display();
    t2108.display();
    t2109.display();
    t2110.display();
    book(btp205, t2110);
    book(btp305, t2108);
    btp105.display();
    btp205.display();
    btp305.display();
    t2108.display();
    t2109.display();
    t2110.display();
    book(btp205, t2108);

    ***** 105 Intro to Programming
    ***** 205 Intro to O-O Prg
    ***** 305 O-O Programming
    T2108 available
    T2109 available
    T2110 available

    ***** 105 Intro to Programming
    T2110 205 Intro to O-O Prg
    T2108 305 O-O Programming
    T2108 O-O Programming
    T2109 available
    T2110 Intro to O-O Programming

```

```
book(btp305, t2109);
btp105.display();
btp205.display();
btp305.display();
t2108.display();
t2109.display();
t2110.display();
}
```

```
***** 105 Intro to Programming
T2108 205 Intro to O-O Prg
T2109 305 O-O Programming
T2108 Intro to O-O Programming
T2109 O-O Programming
T2110 available
```

## EXERCISES

- Complete the Workshop on [Containers](#)
- Read StackOverFlow on [Difference Between Aggregation and Composition](#)
- Read Software Engineering Stack Exchange on [Differences between Association, Aggregation and Composition](#)

 [print this page](#)

[← Previous: Templates](#)

[Top](#) 

[Next: Expressions →](#)

	ICT	Home	Outline	Timeline	
Notes	IPC144	OOP244	Workshops	Assignments	Instructor

Designed by Chris Szalwinski

[Copying From This Site](#)

Last Modified: 11/15/2020 19:33

