# Coupling and Cohesion

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together.

There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

## Cohesion

Cohesion is a measure that defines the **degree of intra-dependability within elements of a module**. **The greater the cohesion, the better is the program design.**

It is a measure of the strength of relationship between the methods and data of a class and some unifying purpose or concept served by that class.

Modules with high cohesion tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.

There are seven types of cohesion, namely –

### Co-incidental cohesion (worst)

It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.

### Logical cohesion

When logically categorized elements are put together into a module, it is called logical cohesion.

### Temporal Cohesion

When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.

### Procedural cohesion

When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.

### Communicational cohesion

When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.

### Sequential cohesion

When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.

### Functional cohesion (best)

It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

### Perfect cohesion (atomic)

When elements of a module cannot be reduces any more.

First two types of cohesion are inferior; communicational and sequential cohesion are very good; and functional cohesion is superior.

While functional cohesion is considered the most desirable type of cohesion for a software module, it may not be achievable. There are cases where communicational cohesion is the highest level of cohesion that can be attained under the circumstances.

## Coupling

Coupling is a measure that defines the level of **inter-dependability among modules of a program**. It tells at what level the modules interfere and interact with each other. **The lower the coupling, the better the program**.

Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa.

Low coupling is often thought to be a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.

There are five levels of coupling, namely -

### Procedural programming

### Content coupling (high)

When a module can directly access or modify or refer to the content of another module, it is called content level coupling.

### Common coupling

When multiple modules have read and write access to some global data, it is called common or global coupling.

### Control coupling

Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.

### Stamp coupling (data-structured coupling)

When multiple modules share common data structure and work on different part of it, it is called stamp coupling.

### Data coupling

Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

For more detail of type of coupling check Wikipedia page.

# ****Relationships between objects****

There are many different kinds of relationships two objects may have in real-life, and we use specific “relation type” words to describe these relationships.

For example:

A square “is-a” shape.

A car “has-a” steering wheel.

A computer programmer “uses-a” keyboard.

A flower “depends-on” a bee for pollination.

A student is a “member-of” a class. And

Your brain exists as “part-of” you (at least, we can reasonably assume so if you’ve gotten this far).

All of these relation types have useful analogies in C++.

We’ll explore the nuances of the relation types

**“part-of”,**

**“has-a”,**

**“uses-a”,**

**“depends-on”,** and

**“member-of”,**

and show how they can be useful in the context of C++ classes. We’ll also explore a couple of related topics that don’t fit nicely anywhere else.

# Object composition

In real-life, complex objects are often built from smaller, simpler objects. For example, A personal computer is built from a CPU, a motherboard, some memory, etc… .

This process of building complex objects from simpler ones is called **object composition.**

Broadly speaking, object composition models a “has-a” relationship between two objects.

A car “has-a” transmission. Your computer “has-a” CPU.

The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component.

When we build classes with data members, we’re essentially constructing a complex object from simpler parts, which is object composition. For this reason, structs and classes are sometimes referred to as **composite types.**

## Types of object composition

There are two basic subtypes of object composition:

* composition
* aggregation

*A note on terminology: the term “composition” is often used to refer to both composition and aggregation, not just to the composition subtype. In this tutorial, we’ll use the term “object composition” when we’re referring to both, and “composition” when we’re referring specifically to the composition subtype.*

# Composition (“part-of”)

To qualify as a composition, an object and a part must have the following relationship:

* The part (member) is part of the object (class)
* The part (member) can only belong to one object (class) at a time
* The part (member) has its existence managed by the object (class)
* The part (member) does not know about the existence of the object (class)

A good real-life example of a composition is the relationship between a person’s body and a heart. Let’s examine these in more detail.

Composition relationships are part-whole relationships where the part must constitute part of the whole object. For example, a heart is a part of a person’s body. The part in a composition can only be part of one object at a time. A heart that is part of one person’s body cannot be part of someone else’s body at the same time.

In a composition relationship, the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created, and destroyed when the object is destroyed. But more broadly, it means the object manages the part’s lifetime in such a way that the user of the object does not need to get involved. For example, when a body is created, the heart is created too. When a person’s body is destroyed, their heart is destroyed too. Because of this, composition is sometimes called a **“death relationship”**.

We call this a **unidirectional relationship**, because the body knows about the heart, but not the other way around.

While object composition models has-a type relationships (a body has-a heart, a fraction has-a denominator), we can be more precise and say that composition models “part-of” relationships (a heart is part-of a body, a numerator is part of a fraction). Composition is often used to model physical relationships, where one object is physically contained inside another.

The parts of a composition can be singular or multiplicative -- for example, a heart is a singular part of the body, but a body contains 10 fingers (which could be modeled as an array).

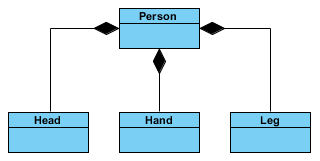
## Implementing compositions

Compositions are one of the easiest relationship types to implement in C++. They are typically created as structs or classes with normal data members. Because these data members exist directly as part of the struct/class, their lifetimes are bound to that of the class instance itself.

Compositions that need to do dynamic allocation or deallocation may be implemented using pointer data members. In this case, the composition class should be responsible for doing all necessary memory management itself (not the user of the class).

In general, if you can design a class using composition, you should design a class using composition. Classes designed using composition are straightforward, flexible, and robust (in that they clean up after themselves nicely).

## UML Diagram



UML class diagram showing composition

## Examples

Many games and simulations have creatures or objects that move around a board, map, or screen. One thing that all of these creatures/objects have in common is that they all have a location. In this example, we are going to create a creature class that uses a point class to hold the creature’s location.

First, let’s design the point class. Our creature is going to live in a 2d world, so our point class will have 2 dimensions, X and Y. We will assume the world is made up of discrete squares, so these dimensions will always be integers.

Point2D.h:

#ifndef POINT2D\_H

#define POINT2D\_H

#include <iostream>

class Point2D {

private:

int m\_x;

int m\_y;

public:

// A default constructor

Point2D() : m\_x{ 0 }, m\_y{ 0 } { }

// A specific constructor

Point2D(int x, int y) : m\_x{ x }, m\_y{ y } { }

// An overloaded output operator

friend std::ostream& operator<<(std::ostream& out, const Point2D& point) {

out << '(' << point.m\_x << ", " << point.m\_y << ')';

return out;

}

// Access functions

void setPoint(int x, int y) {

m\_x = x;

m\_y = y;

}

};

#endif

Note that because we’ve implemented all of our functions in the header file (for the sake of keeping the example concise), there is no Point2D.cpp.

This Point2d class is a composition of its parts: location values x and y are part-of Point2D, and their lifespan is tied to that of a given Point2D instance.

Now let’s design our Creature. Our Creature is going to have a few properties: a name, which will be a string, and a location, which will be our Point2D class.

Creature.h:

#ifndef CREATURE\_H

#define CREATURE\_H

#include <iostream>

#include <string>

#include "Point2D.h"

class Creature {

private:

std::string m\_name;

Point2D m\_location;

public:

Creature(const std::string& name, const Point2D& location)

: m\_name{ name }, m\_location{ location } {

}

friend std::ostream& operator<<(std::ostream& out, const Creature& creature) {

out << creature.m\_name << " is at " << creature.m\_location;

return out;

}

void moveTo(int x, int y) {

m\_location.setPoint(x, y);

}

};

#endif

This Creature is also a composition of its parts. The creature’s name and location have one parent, and their lifetime is tied to that of the Creature they are part of.

And finally, Main.cpp:

#include <string>

#include <iostream>

#include "Creature.h"

#include "Point2D.h"

int main() {

std::cout << "Enter a name for your creature: ";

std::string name;

std::cin >> name;

Creature creature{ name, { 4, 7 } };

while (true) {

// print the creature's name and location

std::cout << creature << '\n';

std::cout << "Enter new X location for creature (-1 to quit): ";

int x{ 0 };

std::cin >> x;

if (x == -1)

break;

std::cout << "Enter new Y location for creature (-1 to quit): ";

int y{ 0 };

std::cin >> y;

if (y == -1)

break;

creature.moveTo(x, y);

}

return 0;

}

Here’s a transcript of this code being run:

Enter a name for your creature: Marvin

Marvin is at (4, 7)

Enter new X location for creature (-1 to quit): 6

Enter new Y location for creature (-1 to quit): 12

Marvin is at (6, 12)

Enter new X location for creature (-1 to quit): 3

Enter new Y location for creature (-1 to quit): 2

Marvin is at (3, 2)

Enter new X location for creature (-1 to quit): -1

## Variants on the composition theme

Although most compositions directly create their parts when the composition is created and directly destroy their parts when the composition is destroyed, there are some variations of composition that bend these rules a bit.

For example

A composition may defer creation of some parts until they are needed. For example, a string class may not create a dynamic array of characters until the user assigns the string some data to hold.

A composition may opt to use a part that has been given to it as input rather than create the part itself.

A composition may delegate destruction of its parts to some other object (e.g. to a garbage collection routine).

The key point here is that the composition should manage its parts without the user of the composition needing to manage anything.

## Composition and subclasses

One question that new programmers often ask when it comes to object composition is, “When should I use a subclass instead of direct implementation of a feature?”. For example, instead of using the Point2D class to implement the Creature’s location, we could have instead just added 2 integers to the Creature class and written code in the Creature class to handle the positioning. However, making Point2D its own class has a number of benefits:

Each individual class can be kept relatively simple and straightforward, focused on performing one task well. This makes those classes easier to write and much easier to understand, as they are more focused. For example, Point2D only worries about point-related stuff, which helps keep it simple.

Each subclass can be self-contained, which makes them reusable. For example, we could reuse our Point2D class in a completely different application. Or if our creature ever needed another point (for example, a destination it was trying to get to), we can simply add another Point2D member variable.

The parent class can have the subclasses do most of the hard work, and instead focus on coordinating the data flow between the subclasses. This helps lower the overall complexity of the parent object, because it can delegate tasks to its children, who already know how to do those tasks. For example, when we move our Creature, it delegates that task to the Point class, which already understands how to set a point. Thus, the Creature class does not have to worry about how such things would be implemented.

**Tip**

A good rule of thumb is that each class should be built to accomplish a single task. That task should either be the storage and manipulation of some kind of data (e.g. Point2D, std::string), OR the coordination of subclasses (e.g. Creature). Ideally not both.

In this case of our example, it makes sense that Creature shouldn’t have to worry about how Points are implemented, or how the name is being stored. Creature’s job isn’t to know those intimate details. Creature’s job is to worry about how to coordinate the data flow and ensure that each of the subclasses knows what it is supposed to do. It’s up to the individual subclasses to worry about how they will do it.

# Aggregation (“has-a”)

To qualify as an aggregation, a whole object and its parts must have the following relationship:

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

Like a composition, an aggregation is still a part-whole relationship, where the parts are contained within the whole, and it is a unidirectional relationship.

However, unlike a composition, parts can belong to more than one object at a time, and the whole object is not responsible for the existence and lifespan of the parts. When an aggregation is created, the aggregation is not responsible for creating the parts. When an aggregation is destroyed, the aggregation is not responsible for destroying the parts.

For example, consider the relationship between a person and their home address. In this example, for simplicity, we’ll say every person has an address. However, that address can belong to more than one person at a time: for example, to both you and your roommate or significant other. However, that address isn’t managed by the person -- the address probably existed before the person got there, and will exist after the person is gone. Additionally, a person knows what address they live at, but the addresses don’t know what people live there. Therefore, this is an aggregate relationship.

We can say that aggregation models “has-a” relationships (a department has teachers, the car has an engine).

Similar to a composition, the parts of an aggregation can be singular or multiplicative.

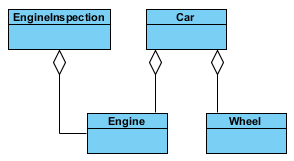
## Implementing aggregations

Because aggregations are similar to compositions in that they are both part-whole relationships, they are implemented almost identically, and the difference between them is mostly semantic. In a composition, we typically add our parts to the composition using normal member variables (or pointers where the allocation and deallocation process is handled by the composition class).

In an aggregation, we also add parts 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 aggregation 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 parts exist outside of the scope of the class, when the class is destroyed, the pointer or reference member variable will be destroyed (but not deleted). Consequently, the parts themselves will still exist.

## UML Diagram



UML class diagram showing aggregation.

## Example

Let’s take a look at a Teacher and Department example in more detail. In this example, we’re going to make a couple of simplifications: First, the department will only hold one teacher. Second, the teacher will be unaware of what department they’re part of.

#include <iostream>

#include <string>

class Teacher {

private:

std::string m\_name{};

public:

Teacher(const std::string& name) : m\_name{ name } { }

const std::string& getName() const { return m\_name; }

};

class Department {

private:

const Teacher& m\_teacher; // This dept holds only one teacher for simplicity, but it could hold many teachers

public:

Department(const Teacher& teacher) : m\_teacher{ teacher } { }

};

int main() {

// Create a teacher outside the scope of the Department

Teacher bob{ "Bob" }; // create a teacher

{

// Create a department and use the constructor parameter to pass the teacher to it.

Department department{ bob };

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

// bob still exists here, but the department doesn't

std::cout << bob.getName() << " still exists!\n";

return 0;

}

In this case, bob is created independently of department, and then passed into department‘s constructor. When department is destroyed, the m\_teacher reference is destroyed, but the teacher itself is not destroyed, so it still exists until it is independently destroyed later in main().

## Pick the right relationship for what you’re modeling

Although it might seem a little silly in the above example that the Teachers don’t know what Department they’re working for, that may be totally fine in the context of a given program. When you’re determining what kind of relationship to implement, implement the simplest relationship that meets your needs, not the one that seems like it would fit best in a real-life context.

For example, if you’re writing a body shop simulator, you may want to implement a car and engine as an aggregation, so the engine can be removed and put on a shelf somewhere for later. However, if you’re writing a racing simulation, you may want to implement a car and an engine as a composition, since the engine will never exist outside of the car in that context.

**Best practice**

Implement the simplest relationship type that meets the needs of your program, not what seems right in real-life.

# Summarizing composition and aggregation

## Compositions

* Typically use normal member variables
* Can use pointer members if the class handles object allocation/deallocation itself
* Responsible for creation/destruction of parts

## Aggregations

* Typically use pointer or reference members that point to or reference objects that live outside the scope of the aggregate class
* Not responsible for creating/destroying parts

It is worth noting that the concepts of composition and aggregation 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 parts 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 aggregation, and created/destroyed independently.

While aggregations can be extremely useful, they are also potentially more dangerous, because aggregations do not handle deallocation of their parts. Deallocations are left to an external party to do. If the external party no longer has a pointer or reference to the abandoned parts, or if it simply forgets to do the cleanup (assuming the class will handle that), then memory will be leaked.

For this reason, compositions should be favored over aggregations.

# Association (“uses-a”)

Let's 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.

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

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.

## UML Diagram

1. A single student can associate with multiple teachers:



1. The example indicates that every Instructor has one or more Students:



1. We can also indicate the behaviour of an object in an association (i.e., the role of an object) using role names.



## Example

#include <functional> // reference\_wrapper

#include <iostream>

#include <string>

#include <vector>

// Since Doctor and Patient have a circular dependency, we're going to forward declare Patient

class Patient;

class Doctor {

private:

std::string m\_name{};

std::vector<std::reference\_wrapper<const Patient>> m\_patient{};

public:

Doctor(const std::string& name) : m\_name{ name } { }

void addPatient(Patient& patient);

// We'll implement this function below Patient since we need Patient to be defined at that point

friend std::ostream& operator<<(std::ostream& out, const Doctor& doctor);

const std::string& getName() const { return m\_name; }

};

class Patient {

private:

std::string m\_name{};

std::vector<std::reference\_wrapper<const 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

void addDoctor(const Doctor& doctor) {

m\_doctor.push\_back(doctor);

}

public:

Patient(const 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& patient);

const std::string& getName() const { return m\_name; }

// We'll friend Doctor::addPatient() so it can access the private function Patient::addDoctor()

friend void Doctor::addPatient(Patient& patient);

};

void Doctor::addPatient(Patient& patient) {

// Our doctor will add this patient

m\_patient.push\_back(patient);

// and the patient will also add this doctor

patient.addDoctor(\*this);

}

std::ostream& operator<<(std::ostream& out, const Doctor& doctor) {

if (doctor.m\_patient.empty()) {

out << doctor.m\_name << " has no patients right now";

return out;

}

out << doctor.m\_name << " is seeing patients: ";

for (const auto& patient : doctor.m\_patient)

out << patient.get().getName() << ' ';

return out;

}

std::ostream& operator<<(std::ostream& out, const Patient& patient) {

if (patient.m\_doctor.empty()) {

out << patient.getName() << " has no doctors right now";

return out;

}

out << patient.m\_name << " is seeing doctors: ";

for (const auto& doctor : patient.m\_doctor)

out << doctor.get().getName() << ' ';

return out;

}

int main() {

// Create a Patient outside the scope of the Doctor

Patient dave{ "Dave" };

Patient frank{ "Frank" };

Patient betsy{ "Betsy" };

Doctor james{ "James" };

Doctor scott{ "Scott" };

james.addPatient(dave);

scott.addPatient(dave);

scott.addPatient(betsy);

std::cout << james << '\n';

std::cout << scott << '\n';

std::cout << dave << '\n';

std::cout << frank << '\n';

std::cout << betsy << '\n';

return 0;

}

Output:

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:

## Example

#include <string>

class Course {

private:

std::string m\_name;

const Course\* m\_prerequisite;

public:

Course(const std::string& name, const 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 previous cases, we’ve used either pointers or references 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 or reference member:

## Example

#include <iostream>

#include <string>

class Car {

private:

std::string m\_name;

int m\_id;

public:

Car(const std::string& name, int id)

: m\_name{ name }, m\_id{ id } { }

const std::string& getName() const { return m\_name; }

int getId() const { 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 create 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]{ { "Prius", 4 }, { "Corolla", 17 }, { "Accord", 84 }, { "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(const std::string& name, int carId)

: m\_name{ name }, m\_carId{ carId } { }

const std::string& getName() const { return m\_name; }

int getCarId() const { 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:

Association

Aggregation  
  
  
Composition

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

# References

<https://en.wikipedia.org/wiki/Cohesion_(computer_science)>

<https://en.wikipedia.org/wiki/Coupling_(computer_programming)>

<https://www.tutorialspoint.com/software_engineering/software_design_basics.htm>

Chapter 16 - An Introduction to Object Relationships | <https://www.learncpp.com/>

# END