10.5 — Dependencies

BY ALEX ON AUGUST 23RD, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

So far, we've explored 3 types of relationships: composition, aggregation, and association. We've saved the simplest one for last: dependencies.

In casual conversation, we use the term dependency to indicate that an object is reliant upon another object for a given task. For example, if you break your foot, you are dependent on crutches to get around (but not otherwise). Flowers are dependent upon bees to pollinate them, in order to grow fruit or propagate (but not otherwise).

A **dependency** occurs when one object invokes another object's functionality in order to accomplish some specific task. This is a weaker relationship than an association, but still, any change to object being depended upon may break functionality in the (dependent) caller. A dependency is always a unidirectional relationship.

A good example of a dependency that you've already seen many times is std::cout (of type std::ostream). Our classes that use std::cout use it in order to accomplish the task of printing something to the console, but not otherwise.

For example:

```
#include <iostream>

class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    friend std::ostream& operator<< (std::ostream &out, const Point &point);
};

std::ostream& operator<< (std::ostream &out, const Point &point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members directly
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";

    return out;
}

int main()
{
    Point point1(2.0, 3.0, 4.0);

    std::cout << point1;

    return 0;
}
```

In the above code, Point isn't directly related to std::cout, but it has a dependency on std::cout since operator<< uses std::cout to print the Point to the console.

Dependencies vs Association in C++

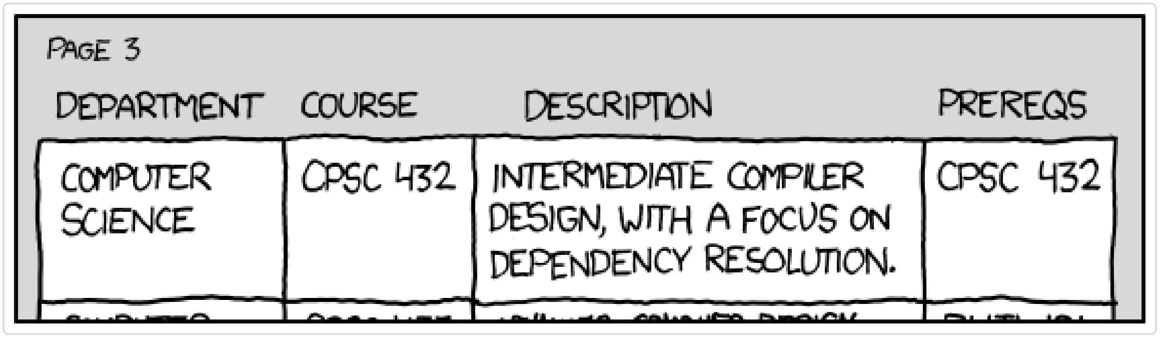
There's typically some confusion about what differentiates a dependency from an association.

In C++, associations are a relationship between two classes at the class level. That is, one class keeps a direct or indirect "link" to the associated class as a member. For example, a Doctor class has an array of pointers to its Patients as a member. You can always ask the Doctor who its patients are. The Driver class holds the id of the Car the driver object owns as an integer member. The Driver always knows what Car is associated with it.

Dependencies typically are not represented at the class level -- that is, the object being depended on is not linked as a member. Rather, the object being depended on is typically instantiated as needed (like opening a file to write data to), or passed into a function as a parameter (like std::ostream in the overloaded operator<< above).

Humor break

Dependencies (courtesy of our friends at **xkcd**):



Of course, you and I know that this is actually a reflexive association!

10.6 -- Container classes

Index

10.4 -- Association

C++ TUTORIAL | PRINT THIS POST

18 comments to 10.5 — Dependencies

Michael Stef
October 13, 2018 at 1:01 am · Reply

in the section: "Dependencies vs Association in C++"
Since the dependent is the caller, I'am not sure but should it be "object we depend on" instead "dependent object" ?

Alex
October 15, 2018 at 9:31 am · Reply

I think you're correct. Lesson updated. Thanks!

stannis
May 5, 2018 at 1:39 pm · Reply

"A dependency is always a unidirectional relationship."
Why can't you have bidirectional dependency?
Eg. If a flower depends on bees to pollinate and bees depend on flower to feed, is this not bidirectional dependency?

Alex
May 6, 2018 at 9:38 pm · Reply

C++ can't easily represent bidirectional dependencies, so in such a case we'd model this as mutual dependencies (flower has a dependency on bee, and bee has a dependency on flower).

Benjamin
January 12, 2018 at 2:03 am · Reply

Hi!

Probably this is more a question about English language (which is not my mother tongue) than C++, but I was surprised when you wrote "any change to the dependent object may break functionality in the caller". Is not the caller the dependent one, as he depends on the functionality of the called object, which is used in the dependency-relationship?

In the example with the flowers and the bees (hehe), you actually call the flower the dependent one. This agrees with my understanding. The two statements are not contradicting in their use of "dependent"?

PS.: Little typo: the last sentence "Of course, you and I know that this actually a reflexive association!" misses an "is".

Alex
January 16, 2018 at 11:41 am · Reply

You are correct, I did not write what I had intended. The intention was to note that a change to the object being depended upon could break the class with the dependency. I've updated the text, as well as corrected the typo.

Thanks!

DOG_TRAINER
November 26, 2017 at 8:14 pm · Reply

I have been reading this far, but I still don't get otherwise unrelated and not otherwise mean.

Alex
November 28, 2017 at 5:49 pm · Reply

"but not otherwise" means "but not in situations where the preceding statement isn't true". For example, if I say I like eggs on Sunday but not otherwise, that means I only like eggs on Sunday and not on other days of the week.

Simon
June 7, 2017 at 12:13 am · Reply

Alex I feel these relations are really important for finding better solutions for real life problems and I'm really interested in them. What is a good book you might want to suggest for reading about them in more depth?

Alex
June 7, 2017 at 2:12 pm · Reply

I don't have any good books in mind for this -- I scraped these together from various sources on the web. You might try doing some reading on UML modelling.

ali
December 17, 2016 at 11:38 am · Reply

hello Alex I am a newly graduated student when I was in college we have something about DBMS (database management system) the thing is we studied the relationships between entities like (the department has one manager, the company has several departments) and did some kind of query using SQL the quistion: is the relationships in SQL are implemented just like you explained? if not could we do like them in c++ techniques

Alex
December 19, 2016 at 10:30 am · Reply

Yes, these relationships are applicable to databases. Databases are often described using UML, and UML contains symbols to model these different relationship types.

I don't talk about UML here specifically, because it's not directly related to C++, but a lot of the material in this chapter maps to topics that are often discussed in UML.

ali
December 20, 2016 at 12:24 am · Reply

thank you ...

daniel
October 10, 2016 at 2:00 pm · Reply

Thank you for this great tutorial.
I found one typo: There should be a verb "IS" in the last sentence.

Of course, you and I know that this "IS" actually a reflexive association!

Matt
September 30, 2016 at 5:26 pm · Reply

Typo in section "Dependencies vs Association in C++"... second paragraph, second sentence. I think "a a" should be "as a".

Alex
September 30, 2016 at 8:20 pm · Reply

Indeed. Thanks again.

daniel
August 24, 2016 at 4:37 pm · Reply

when you say

```
std::ostream& operator<< (std::ostream &out, const Point &point)
```

outside of the class and don't mention Point::std::ostream, how does compiler know its the operator overload for point class.
what if there is more than one class and both have this kind of overload ?

Alex
August 25, 2016 at 11:39 am · Reply

There is no such thing as Point::std::ostream. I'll presume you meant Point::operator<<.
What happens is when you do something like this:

```
std::cout << p; // where p is a Point
```

C++ tries to match this to a function. It'll try to see if there is a std::ostream::operator<<(Point) function, but there obviously won't be. Next it checks for an operator<<(std::ostream, Point). It will find our overloaded operator definition and match to that. Note that in a binary operator, the left hand side operand always becomes the implicit object, so there's never any ambiguity about which class to look in. For non-member functions, the left hand side operand becomes the first parameter and the right hand side operator becomes the second, so there's also no ambiguity here.