

Re: Aggregation vs composition

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2004-04/1393.html>

From: Matthias Hofmann (hofmann_at_anvil-soft.com)

Date: 04/27/04

Date: Tue, 27 Apr 2004 18:14:58 +0200

Daniel T. <postmaster@earthlink.net> schrieb in im Newsbeitrag:
postmaster-48E91C.08222327042004@news5.west.earthlink.net...

> "Ilja Preuß" <preuss@disy.net> wrote:

>

> > Daniel T. wrote:

> >

> > >>>>>>>

> > >>>>>>> |A|<#>----->|B|<-----|C|

> > >>>>>>>

> > > Humm... If containment is being defined as "having life time
> > > responsibility for" then why would you use it when the whole doesn't
> > > have lifetime responsibility for its parts? If, in the diagram above,
> > > an A and C object both hold a reference to a B object, then the A
> > > object decidedly does **not** have lifetime control: A and C **share**
> > > that control.

> >

> > Control != Responsibility. A is responsible for controlling the
lifetime, C

> > is not.

>

> However in the diagram above, C is just as responsible as A in managing
> the lifetime of the B objects they share. If two objects share a third,
> then they also share responsibility for the disposition of that third
> object. I'm not sure why I'm having such a hard time getting this across.

Obviously your concern is the problem of dangling references and wild pointers, which is indeed something a programmer has to be aware of. Nevertheless, holding a reference does not mean **controlling** lifetime, it rather means **depending on** lifetime. In your A-B-C example, the C object is NOT responsible for the disposition of the B object.

>

> Yes, A, if it knows all of the clients of a particular B object, could
> request that all those clients drop their reference, then drop its
> reference (thus "destroying" the object,) but being able to request that
> others stop using the object is not "responsibility for the disposition"
> of said object, if any one client refuses then the object being held

comp.object: Re: Aggregation vs composition

> *doesn't get destroyed.*

If any of the clients refuses, then it's his own fault, as he will end up with a dead reference. A is not asking for permission to destroy B, it merely informs C that B is about to go up in smoke, no matter whether C wants or not.

>

> *Now I realize the above is only true in GC languages, in C++, A can
> explicitly destroy a B object at any moment, the other clients of that B
> object be damned. (Of course this applies to every client of that B
> object.) First, if the composition relationship only matters in C++,
> then how is it language neutral? Second, A's ability to make such
> imperious demands on objects that it doesn't have responsibility for
> (namely other clients of B,) smacks of god class problems and IMO is
> irresponsible.*

This problem is not created by the UML. The UML only makes implementers aware of such things, and that's the gooo thing about the UML being able to show such situations.

>

> *Maybe we will just have to agree to disagree, but I ask this: If I'm
> wrong, look at your code and show me the difference between Composition
> and Aggregation.*

Take a look at my other post.

> *Is it because (as someone has already said) that there
> is a line "delete child;" somewhere in the class? I dare say that that
> line exists *somewhere* or you have a leak. That would mean that *all*
> objects have a Containment relationship with *something*. The*

No, you only need to delete objects that are allocated dynamically. Use a regular data member and you have composition also:

```
class Part {};
```

```
class Whole  
{  
  Part part;  
};
```

> *Containment relationship should be the most used relationship in UML if
> that were the case. Is it because some object happens to be the last one
> to hold a reference to some other, thus when that reference is dropped
> the object is gone? Again, this could be said of every object in the
> system...*

Reference counting not always present. Objects do not always keep track of how many other objects hold references to them. Often, it is a matter of

comp.object: Re: Aggregation vs composition

timing – you usually know when an object goes out of scope, as the rules on that are very clear (at least in C++). You simply don't use an object after it has been destroyed.

Best regards,

Matthias