Inheritance vs. Aggregation [closed]

Asked 16 years, 1 month ago Modified 12 years, 3 months ago

Viewed 121k times

155

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

Closed 12 years ago.

There are two schools of thought on how to best extend, enhance, and reuse code in an object-oriented system:

1. Inheritance: extend the functionality of a class by creating a subclass. Override superclass members in the subclasses to provide new functionality. Make methods abstract/virtual to force subclasses to "fill-in-the-blanks" when the superclass wants a particular interface but is agnostic about its implementation.
2. Aggregation: create new functionality by taking other classes and combining them into a new class. Attach an common interface to this new class for interoperability with other code.

What are the benefits, costs, and consequences of each? Are there other alternatives?

I see this debate come up on a regular basis, but I don't think it's been asked on Stack Overflow yet (though there is some related discussion). There's also a surprising lack of good Google results for it.

oop inheritance language-agnostic aggregation

Share

Improve this question

Follow

edited Aug 30, 2012 at 19:27

casperOne

74.5k ● 19 ● 188 ● 259

asked Nov 6, 2008 at 17:16

Craig Walker

51.6k ● 57 ● 155 ● 212

1 Related: Prefer composition over inheritance? – Melebius May 13, 2020 at 8:00

12 Answers

Sorted by: Highest score (default)

204

It's not a matter of which is the best, but of when to use what.

In the 'normal' cases a simple question is enough to find out if we need inheritance or aggregation.

- If The new class **is** more or less as the original class. Use inheritance. The new class is now a subclass of the original class.
- If the new class must **have** the original class. Use aggregation. The new class has now the original class as a member.

However, there is a big gray area. So we need several other tricks.

- If we have used inheritance (or we plan to use it) but we only use part of the interface, or we are forced to override a lot of functionality to keep the correlation logical. Then we have a big nasty smell that indicates that we had to use aggregation.
- If we have used aggregation (or we plan to use it) but we find out we need to copy almost all of the functionality. Then we have a smell that points in the direction of inheritance.

To cut it short. We should use aggregation if part of the interface is not used or has to be changed to avoid an illogical situation. We only need to use inheritance, if we need almost all of the functionality without major changes. And when in doubt, use Aggregation.

An other possibility for, the case that we have an class that needs part of the functionality of the original class, is to split the original class in a root class and a sub class. And let the new class inherit from the root class. But you should take care with this, not to create an illogical separation.

Lets add an example. We have a class 'Dog' with methods: 'Eat', 'Walk', 'Bark', 'Play'.

```
class Dog
  Eat;
  Walk;
  Bark;
  Play;
end;
```

We now need a class 'Cat', that needs 'Eat', 'Walk', 'Purr', and 'Play'. So first try to extend it from a Dog.

```
class Cat is Dog
  Purr;
end;
```

Looks, alright, but wait. This cat can Bark (Cat lovers will kill me for that). And a barking cat violates the principles of the universe. So we need to override the Bark method so that it does nothing.

```
class Cat is Dog
  Purr;
  Bark = null;
end;
```

Ok, this works, but it smells bad. So lets try an aggregation:

```
class Cat
  has Dog;
  Eat = Dog.Eat;
  Walk = Dog.Walk;
  Play = Dog.Play;
  Purr;
end;
```

Ok, this is nice. This cat does not bark anymore, not even silent. But still it has an internal dog that wants out. So lets try solution number three:

```
class Pet
  Eat;
  Walk;
  Play;
end;

class Dog is Pet
  Bark;
end;

class Cat is Pet
  Purr;
end;
```

This is much cleaner. No internal dogs. And cats and dogs are at the same level. We can even introduce other pets to extend the model. Unless it is a fish, or something that does not walk. In that case we again need to refactor. But that is something for an other time.

Share Improve this answer Follow

edited Sep 25, 2011 at 8:31

answered Nov 6, 2008 at 17:28

Toon Krijthe
53.4k ● 38 ● 149 ● 202

6 The "reuse almost all functionality of a class" is the one time I do actually prefer inheritance. What I'd *really* like is a language that has the ability to easily say "delegate to this aggregated object for these specific methods"; that's the best of both worlds. – Craig Walker Nov 6, 2008 at 19:45

I'm not sure about other languages, but Delphi has a mechanism, that let members implement part of the interface. – Toon Krijthe Nov 6, 2008 at 20:14

2 Objective C uses Protocols that let you decide whether your function is required or optional. – cantfindaname88 Jan 20, 2013 at 3:29

3 Great answer with a practical example. I would design the Pet class with a method called `makeSound` and let each subclass implement their own kind of sound. This will then help in situations where you have many pets and just do `for_each pet in the list of pets, pet.makeSound`. – blongho Aug 26, 2019 at 5:42

What if my pet is a goldfish? – KevDog Apr 17, 2022 at 23:32

42

At the beginning of GOF they state

> Favor object composition over class inheritance.

This is further discussed here

Share Improve this answer edited Sep 27, 2011 at 20:50

Follow

Toon Krijthe
53.4k ● 38 ● 149 ● 202

answered Nov 6, 2008 at 17:28

toolkit
50.2k ● 18 ● 111 ● 138

33

The difference is typically expressed as the difference between "is a" and "has a". Inheritance, the "is a" relationship, is summed up nicely in the Liskov Substitution Principle. Aggregation, the "has a" relationship, is just that - it shows that the aggregating object *has* one of the aggregated objects.

Further distinctions exist as well - private inheritance in C++ indicates a "is implemented in terms of" relationship, which can also be modeled by the aggregation of (non-exposed) member objects as well.

Share Improve this answer

Follow

answered Nov 6, 2008 at 17:25

Harper Shelby
16.6k ● 2 ● 46 ● 52

1 Also, difference is summed up nicely in the very question that you are supposed to respond to. I would like that people first read questions before start translating. Do yo blindly promote design patters to become a member of the elite club? – Val Jan 17, 2013 at 19:19

I do not like this approach, it's more about composition – Alireza Rahmani Khalili Apr 1, 2018 at 12:19

16

Here's my most common argument:

In any object-oriented system, there are two parts to any class:

1. Its *interface*: the "public face" of the object. This is the set of capabilities it announces to the rest of the world. In a lot of languages, the set is well defined into a "class". Usually these are the method signatures of the object, though it varies a bit by language.
2. Its *implementation*: the "behind the scenes" work that the object does to satisfy its interface and provide functionality. This is typically the code and member data of the object.

One of the fundamental principles of OOP is that the implementation is *encapsulated* (ie:hidden) within the class; the only thing that outsiders should see is the interface.

When a subclass inherits from a subclass, it typically inherits *both* the implementation and the interface. This, in turn, means that you're *forced* to accept both as constraints on your class.

With aggregation, you get to choose either implementation or interface, or both -- but you're not forced into either. The functionality of an object is left up to the object itself. It can defer to other objects as it likes, but it's ultimately responsible for itself. In my experience, this leads to a more flexible system: one that's easier to modify.

So, whenever I'm developing object-oriented software, I almost always prefer aggregation over inheritance.

Share Improve this answer Follow

answered Nov 6, 2008 at 17:33

Craig Walker
51.6k ● 57 ● 155 ● 212

I would think that you use an abstract class to define an interface and then use direct inheritance for each concrete implementation class (making it pretty shallow). Any aggregation is under the concrete implementation. – orcmid Nov 6, 2008 at 23:55

If you need additional interfaces than that, return them via methods on the interface of the main-level abstracted interface, rinse repeat. – orcmid Nov 6, 2008 at 23:57

Do you think, aggregation is better in this scenario? A book is a SellingItem, A DigitalDisc is a SelliingItem codereview.stackexchange.com/questions/14077/… – LCJ Aug 1, 2012 at 11:24

13

I gave an answer to "Is a" vs "Has a" : which one is better?.

Basically I agree with other folks: use inheritance only if your derived class truly *is* the type you're extending, not merely because it *contains* the same data. Remember that inheritance means the subclass gains the methods as well as the data.

Does it make sense for your derived class to have all the methods of the superclass? Or do you just quietly promise yourself that those methods should be ignored in the derived class? Or do you find yourself overriding methods from the superclass, making them no-ops so no one calls them inadvertently? Or giving hints to your API doc generation tool to omit the method from the doc?

Those are strong clues that aggregation is the better choice in that case.

Share Improve this answer Follow

edited May 23, 2017 at 12:10
Community Bot
1 ● 1

answered Nov 6, 2008 at 17:48
Bill Karwin
561k ● 87 ● 698 ● 854

I see a lot of "is-a vs. has-a; they're conceptually different" responses on this and the related questions.

6

The one thing I've found in my experience is that trying to determine whether a relationship is "is-a" or "has-a" is bound to fail. Even if you can correctly make that determination for the objects now, changing requirements mean that you'll probably be wrong at some point in the future.

Another thing I've found is that it's *very* hard to convert from inheritance to aggregation once there's a lot of code written around an inheritance hierarchy. Just switching from a superclass to an interface means changing nearly every subclass in the system.

And, as I mentioned elsewhere in this post, aggregation tends to be less flexible than inheritance.

So, you have a perfect storm of arguments against inheritance whenever you have to choose one or the other:

1. Your choice will likely be the wrong one at some point
2. Changing that choice is difficult once you've made it.
3. Inheritance tends to be a worse choice as it's more constraining.

Thus, I tend to choose aggregation -- even when there appears to be a strong is-a relationship.

Share Improve this answer Follow

answered Nov 6, 2008 at 18:12

Craig Walker

51.6k ● 57 ● 155 ● 212

Evolving requirements means your OO design will inevitably be wrong. Inheritance vs. aggregation is just the tip of that iceberg. You can't architect for all possible futures, so just go with the XP idea: solve today's requirements and accept that you may have to refactor. – Bill Karwin Nov 6, 2008 at 18:19

1 I like this line of enquiry and questioning. Concerned about blurrung is-a, has-a, and uses-a though. I start with interfaces (along with identifying the implemented abstractions I want interfaces to). The additional level of indirection is invaluable. Don't follow your aggregation conclusion. – orcmid Nov 6, 2008 at 19:32

That's pretty much my point though. Both inheritance and aggregation are *methods* to solve the problem. One of those methods incurs all sorts of penalties when you have to solve problems tomorrow. – Craig Walker Nov 6, 2008 at 19:39

In my mind, aggregation+interfaces are the alternative to inheritance/subclassing; you can't do polymorphism based on aggregation alone. – Craig Walker Nov 6, 2008 at 19:41

3

The question is normally phrased as Composition vs. Inheritance, and it has been asked here before.

Share Improve this answer Follow

edited May 23, 2017 at 10:31

Community Bot
1 ● 1

answered Nov 6, 2008 at 17:31

Bill the Lizard
405k ● 211 ● 572 ● 889

3

I wanted to make this a comment on the original question, but 300 characters bites [;<).

I think we need to be careful. First, there are more flavors than the two rather specific examples made in the question.

Also, I suggest that it is valuable not to confuse the objective with the instrument. One wants to make sure that the chosen technique or methodology supports achievement of the primary objective, but I don't thing out-of-context which-technique-is-best discussion is very useful. It does help to know the pitfalls of the different approaches along with their clear sweet spots.

For example, what are you out to accomplish, what do you have available to start with, and what are the constraints?

Are you creating a component framework, even a special purpose one? Are interfaces separable from implementations in the programming system or is it accomplished by a practice using a different sort of technology? Can you separate the inheritance structure of interfaces (if any) from the inheritance structure of classes that implement them? Is it important to hide the class structure of an implementation from the code that relies on the interfaces the implementation delivers? Are there multiple implementations to be usable at the same time or is the variation more over-time as a consequence of maintenance and enhancememt? This and more needs to be considered before you fixate on a tool or a methodology.

Finally, is it that important to lock distinctions in the abstraction and how you think of it (as in is-a versus has-a) to different features of the OO technology? Perhaps so, if it keeps the conceptual structure consistent and manageable for you and others. But it is wise not to be enslaved by that and the contortions you might end up making. Maybe it is best to stand back a level and not be so rigid (but leave good narration so others can tell what's up). [I look for what makes a particular portion of a program explainable, but some times I go for elegance when there is a bigger win. Not always the best idea.]

I'm an interface purist, and I am drawn to the kinds of problems and approaches where interface purism is appropriate, whether building a Java framework or organizing some COM implementations. That doesn't make it appropriate for everything, not even close to everything, even though I swear by it. (I have a couple of projects that appear to provide serious counter-examples against interface purism, so it will be interesting to see how I manage to cope.)

Share Improve this answer Follow

answered Nov 6, 2008 at 19:24

orcmid

2,638 ● 19 ● 21

I'll cover the where-these-might-apply part. Here's an example of both, in a game scenario. Suppose, there's a

2

game which has different types of soldiers. Each soldier can have a knapsack which can hold different things.

Inheritance here? There's a marine, green beret & a sniper. These are types of soldiers. So, there's a base class Soldier with Marine, Green Beret & Sniper as derived classes

Aggregation here? The knapsack can contain grenades, guns (different types), knife, medikit, etc. A soldier can be equipped with any of these at any given point in time, plus he can also have a bulletproof vest which acts as armor when attacked and his injury decreases to a certain percentage. The soldier class contains an object of bulletproof vest class and the knapsack class which contains references to these items.

Share Improve this answer Follow

answered Nov 6, 2008 at 17:28

Salman Kasbati
342 ● 1 ● 6

2

I think it's not an either/or debate. It's just that:

1. is-a (inheritance) relationships occur less often than has-a (composition) relationships.
2. Inheritance is harder to get right, even when it's appropriate to use it, so due diligence has to be taken because it can break encapsulation, encourage tight coupling by exposing implementation and so forth.

Both have their place, but inheritance is riskier.

Although of course it wouldn't make sense to have a class Shape 'having-a' Point and a Square classes. Here inheritance is due.

People tend to think about inheritance first when trying to design something extensible, that is what's wrong.

Share Improve this answer

Follow

answered Nov 6, 2008 at 17:31

Vinko Vrsalovic

340k ● 55 ● 340 ● 373

1

Favour happens when both candidate qualifies. A and B are options and you favour A. The reason is that composition offers more extension/flexiblity possiblities than generalization. This extension/flexiblity refers mostly to runtime/dynamic flexibility.

The benefit is not immediately visible. To see the benefit you need to wait for the next unexpected change request. So in most cases those sticked to generlalization fails when compared to those who embraced composition(except one obvious case mentioned later). Hence the rule. From a learning point of view if you can implement a dependency injection successfully then you should know which one to favour and when. The rule helps you in making a decision as well; if you are not sure then select composition.

Summary: Composition :The coupling is reduced by just having some smaller things you plug into something bigger, and the bigger object just calls the smaller object back. Generlization: From an API point of view defining that a method can be overridden is a stronger commitment than defining that a method can be called. (very few occassions when Generalization wins). And never forget that with composition you are using inheritance too, from a interface instead of a big class

Share Improve this answer Follow

edited Sep 13, 2012 at 15:27

answered Sep 20, 2011 at 13:28

Blue Clouds
8,123 ● 9 ● 79 ● 126

0

Both approaches are used to solve different problems. You don't always need to aggregate over two or more classes when inheriting from one class.

Sometimes you do have to aggregate a single class because that class is sealed or has otherwise non-virtual members you need to intercept so you create a proxy layer that obviously isn't valid in terms of inheritance but so long as the class you are proxying has an interface you can subscribe to this can work out fairly well.

Share Improve this answer Follow

answered Nov 6, 2008 at 17:28

cfeduke
23.2k ● 10 ● 63 ● 65