Prefer composition over inheritance?

Asked 16 years, 3 months ago Modified 4 months ago

Viewed 507k times

2011

Why prefer *composition* instead of *inheritance*? What trade-offs are there for each approach? And the converse question: when should I choose *inheritance* instead of *composition*?

language-agnostic oop inheritance composition aggregation

Share

Improve this question

Follow

edited Mar 27, 2023 at 21:38

Mateen Ulhaq

27.1k ● 21 ● 117 ● 152

asked Sep 8, 2008 at 1:58

readonly

355k ● 109 ● 206 ● 206

6 See also which class design is better – maccullt Sep 10, 2008 at 2:22

6 in one sentence inheritance is public if you have a public method and you change it it changes the published api. if you have composition and the object composed has changed you don't have to change your published api. – Tomer Ben David Aug 26, 2015 at 11:00

This is a good read on the topic: Medium.com: Composition vs. Inheritance: Pros and Cons – Gabriel Staples Mar 9, 2023 at 6:50

37 Answers

Sorted by: Highest score (default)

1 2 Next

1467

Prefer composition over inheritance as it is more malleable / easy to modify later, but do not use a compose-always approach. With composition, it's easy to change behavior on the fly with Dependency Injection / Setters. Inheritance is more rigid as most languages do not allow you to derive from more than one type. So the goose is more or less cooked once you derive from TypeA.

My acid test for the above is:

- Does TypeB want to expose the complete interface (all public methods no less) of TypeA such that TypeB can be used where TypeA is expected? Indicates **Inheritance**.
  - e.g. A Cessna biplane will expose the complete interface of an airplane, if not more. So that makes it fit to derive from Airplane.
- Does TypeB want only some/part of the behavior exposed by TypeA? Indicates need for **Composition.**

- e.g. A Bird may need only the fly behavior of an Airplane. In this case, it makes sense to extract it out as an interface / class / both and make it a member of both classes.

Update: Just came back to my answer and it seems now that it is incomplete without a specific mention of Barbara Liskov's Liskov Substitution Principle as a test for 'Should I be inheriting from this type?'

Share Improve this answer Follow

edited Oct 10, 2019 at 8:16
Inanc Gumus
27.8k ● 10 ● 87 ● 104

answered Sep 10, 2008 at 3:04
Gishu
137k ● 47 ● 226 ● 311

114 The second example is straight out of the Head First Design Patterns (amazon.com/First-Design-Patterns-Elisabeth-Freeman/dp/…) book :) I would highly recommend that book to anyone who was googling this question. – Jeshurun Jun 22, 2011 at 4:59

4 It's very clear, but it may miss something : "Does TypeB want to expose the complete interface (all public methods no less) of TypeA such that TypeB can be used where TypeA is expected?" But what if this is true, and TypeB also expose the complete interface of TypeC ? And what if TypeC hasn't been modeled yet ? – Tristan Aug 26, 2011 at 12:04

8 You allude to what I think should be the most basic test: "Should this object be usable by code which expects objects of (what would be) the base type". If the answer is yes, the object *must* inherit. If no, then it probably should

not. If I had my druthers, languages would provide a keyword to refer to "this class", and provide a means of defining a class which should behave just like another class, but not be substitutable for it (such a class would have all "this class" references replaced with itself). – supercat Dec 8, 2011 at 16:16

35 @Alexey - the point is 'Can I pass in a Cessna biplane to all clients that expect an airplane without surprising them?'. If yes, then chances are you want inheritance. – Gishu Oct 28, 2012 at 4:18

10 I'm actually struggling to think of any examples where inheritance would have been my answer, I often find aggregation, composition and interfaces result in more elegant solutions. Many of the above examples could possibly be better explained using those approaches... – Stuart Wakefield Nov 23, 2012 at 11:33

578

Think of containment as a **has a** relationship. A car "has an" engine, a person "has a" name, etc.

Think of inheritance as an **is a** relationship. A car "is a" vehicle, a person "is a" mammal, etc.

I take no credit for this approach. I took it straight from the [Second Edition of Code Complete](#) by [Steve McConnell](#), *Section 6.3*.

Share Improve this answer Follow

edited Sep 22, 2020 at 23:32

James Skemp
8,521 ● 9 ● 69 ● 110

answered Sep 8, 2008 at 2:09

Nick Zalutskiy
15.4k ● 7 ● 54 ● 50

137 This is not always a perfect approach, it's simply a good guideline. the Liskov Substitution Principle is much more accurate (fails less). – Bill K Sep 17, 2008 at 0:25

51 "My car has a vehicle." If you consider that as a separate sentence, not in a programming context, that makes absolutely no sense. And that's the whole point of this technique. If it sounds awkward, it is probably wrong. – Nick Zalutskiy Aug 14, 2011 at 18:27

55 @Nick Sure, but "My Car has a VehicleBehavior" makes more sense (I guess your "Vehicle" class could be named "VehicleBehavior"). So you cannot base your decision on "has a" vs "is a" comparision, you have to use LSP, or you will make mistakes – Tristan Aug 26, 2011 at 11:53

46 Instead of "is a" think of "behaves like." Inheritance is about inheriting behavior, not just semantics. – ybakos Mar 31, 2012 at 19:25

13 This doesn't answer the question. The question is "why" not "what". – Nick Bull Oct 20, 2017 at 10:57

285

If you understand the difference, it's easier to explain.

Procedural Code

An example of this is PHP without the use of classes (particularly before PHP5). All logic is encoded in a set of functions. You may include other files containing helper functions and so on and conduct your business logic by passing data around in functions. This can be very hard to manage as the application grows. PHP5 tries to remedy this by offering a more object-oriented design.

Inheritance

This encourages the use of classes. Inheritance is one of the three tenets of OO design (inheritance, polymorphism, encapsulation).

```
class Person {
   String Title;
   String Name;
   Int Age
}

class Employee : Person {
   Int Salary;
   String Title;
}
```

This is inheritance at work. The `Employee` "**is a**" `Person` or inherits from `Person`. All inheritance relationships are "is-a" relationships. `Employee` also shadows the `Title` property from `Person`, meaning `Employee.Title` will return the `Title` for the `Employee` and not the `Person`.

Composition

Composition is favoured over inheritance. To put it very simply you would have:

```
class Person {
   String Title;
   String Name;
   Int Age;

   public Person(String title, String name, String
age) {
      this.Title = title;
      this.Name = name;
      this.Age = age;
   }

}

class Employee {
   Int Salary;
   private Person person;

   public Employee(Person p, Int salary) {
       this.person = p;
       this.Salary = salary;
   }
}

Person johnny = new Person ("Mr.", "John", 25);
Employee john = new Employee (johnny, 50000);
```

Composition is typically "**has a**" or "**uses a**" relationship. Here the `Employee` class has a `Person`. It does not inherit from `Person` but instead gets the `Person` object passed to it, which is why it "has a" Person.

Composition over Inheritance

Now say you want to create a `Manager` type so you end up with:

```
class Manager : Person, Employee {
   ...
}
```

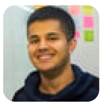
This example will work fine, however, what if `Person` and `Employee` both declared `Title`? Should `Manager.Title` return "Manager of Operations" or "Mr."? Under composition this ambiguity is better handled:

```
Class Manager {
   public string Title;
   public Manager(Person p, Employee e)
   {
      this.Title = e.Title;
   }
}
```

The `Manager` object is composed of an `Employee` and a `Person`. The `Title` behaviour is taken from `Employee`. This explicit composition removes ambiguity among other things and you'll encounter fewer bugs.

Share Improve this answer Follow

edited Sep 22, 2022 at 8:51



Dr.jacky
3,547 ● 6 ● 57 ● 96

answered May 21, 2009 at 7:54

aleemb
32k ● 20 ● 101 ● 114

8 For inheritence: There is no ambiguity. You are implementing the Manager class based on requirements. So you would return "Manager of Operations" if thats what your requirements specified, else you would just use the base

class's implementation. Also you could make Person an abstract class and thereby make sure down-stream classes implement a Title property. – Raj Rao Nov 12, 2010 at 20:21

96 Its important to remember that one might say "Composition over inheritence" but that does not mean "Composition always over Inheritence". "Is a" means inheritence and leads to code reuse. Employee is a Person (Employee does not have a person). – Raj Rao Nov 12, 2010 at 20:26

51 The example is confusing.Employee is a person, so it should use inheritance. You should not use composition for this example, because it is wrong relationship in domain model, even if technically you can declare it in the code. – Michael Freidgeim Jul 24, 2013 at 13:54

34 I disagree with this example. An Employee *is-a* Person, which is a textbook case of proper use of inheritance. I also think that the "issue" the redefinition of the Title field does not make sense. The fact that Employee.Title shadows Person.Title is a sign of poor programming. After all, are "Mr." and "Manager of Operations" really referring to the same aspect of a person (lowercase)? I would rename Employee.Title, and thus be able to reference the Title and JobTitle attributes of an Employee, both of which make sense in real life. Furthermore, there is no reason for Manager (continued...) – Resigned June 2023 Nov 10, 2014 at 0:09

25 (... continued) to inherit from both Person and Employee -- after all, Employee already inherits from Person. In more complex models, where a person might be a Manager and an Agent, it is true that multiple inheritance can be used (carefully!), but it would be preferable in many environments to have an abstract Role class from which Manager (contains Employees s/he manages) and Agent (contains Contracts, and other information) inherit. Then, an Employee is-a Person who has-multiple Roles. Thus, both composition

and inheritance are used properly. – Resigned June 2023
Nov 10, 2014 at 0:13

216

With all the undeniable benefits provided by inheritance, here's some of its disadvantages.

Disadvantages of Inheritance:

1. You can't change the implementation inherited from super classes at runtime (obviously because inheritance is defined at compile time).
2. Inheritance exposes a subclass to details of its parent class implementation, that's why it's often said that inheritance breaks encapsulation (in a sense that you really need to focus on interfaces only not implementation, so reusing by sub classing is not always preferred).
3. The tight coupling provided by inheritance makes the implementation of a subclass very bound up with the implementation of a super class that any change in the parent implementation will force the sub class to change.
4. Excessive reusing by sub-classing can make the inheritance stack very deep and very confusing too.

On the other hand **Object composition** is defined at runtime through objects acquiring references to other objects. In such a case these objects will never be able to reach each-other's protected data (no encapsulation break) and will be forced to respect each other's interface. And in this case also, implementation dependencies will be a lot less than in case of inheritance.

Share Improve this answer Follow

edited Mar 6, 2019 at 11:32

answered May 21, 2009 at 8:34

Ahmed

11.4k ● 16 ● 57 ● 67

13 This is one of the better answers, in my opinion - I will add to this that trying to re-think your problems in terms of composition, in my experience, tends to lead to smaller, simpler, more self-contained, more reusable classes, with a clearer, smaller, more focused scope of responsibility. Often this means there is less need for things like dependency injection or mocking (in tests) as smaller components are usually able to stand on their own. Just my experience. YMMV :-) – mindplay.dk Jan 17, 2014 at 0:51

4 The last paragraph in this post really clicked for me. Thank you. – Salx May 25, 2016 at 19:00

1 Even though you mention a good technique, I don't think you're answering the question. It's not comparing runtime against compile time, but inheritance against composition, which REUSES code from either ONE or MANY classes and can OVERRIDE or ADD new logic. What you describe as object composition, is simply injecting logic via objects assigned as properties in a class, which is a great alternative for runtime manipulation. Thanks! – Isaak Eriksson Jun 24, 2021 at 9:13

Inheritance exposes a subclass to details of its parent class implementation...Is this through protected members?
– AlexBor Jan 30, 2023 at 9:53

114

Another, very pragmatic reason, to prefer composition over inheritance has to do with your domain model, and mapping it to a relational database. It's really hard to map inheritance to the SQL model (you end up with all sorts of hacky workarounds, like creating columns that aren't always used, using views, etc). Some ORMLs try to deal with this, but it always gets complicated quickly. Composition can be easily modeled through a foreign-key relationship between two tables, but inheritance is much harder.

Share Improve this answer Follow

answered Sep 8, 2008 at 2:48
Tim Howland
7,970 ● 4 ● 29 ● 46

106

While in short words I would agree with "Prefer composition over inheritance", very often for me it sounds like "prefer potatoes over coca-cola". There are places for inheritance and places for composition. You need to understand difference, then this question will disappear. What it really means for me is "if you are going to use inheritance - think again, chances are you need composition".

You should prefer potatoes over coca cola when you want to eat, and coca cola over potatoes when you want to drink.

Creating a subclass should mean more than just a convenient way to call superclass methods. You should use inheritance when subclass "is-a" super class both structurally and functionally, when it can be used as superclass and you are going to use that. If it is not the case - it is not inheritance, but something else. Composition is when your objects consists of another, or has some relationship to them.

So for me it looks like if someone does not know if he needs inheritance or composition, the real problem is that he does not know if he want to drink or to eat. Think about your problem domain more, understand it better.

Share Improve this answer Follow

answered May 8, 2009 at 22:29

Pavel Feldman

4,719 ● 7 ● 33 ● 32

6 The right tool for the right job. A hammer may be better at pounding things than a wrench, but that doesn't mean one should view a wrench as "an inferior hammer". Inheritance can be helpful when the things that are added to the subclass are necessary for the object to behave as a superclass object. For example, consider a base class `InternalCombustionEngine` with a derived class `GasolineEngine`. The latter adds things like spark plugs, which the base class lacks, but using the thing as an `InternalCombustionEngine` will cause the spark plugs to get used. – supercat Oct 29, 2012 at 15:07

76

Didn't find a satisfactory answer here, so I wrote a new one.

To understand why "*prefer* composition over inheritance", we need first get back the assumption omitted in this shortened idiom.

There are two benefits of inheritance: subtyping and subclassing

1. **Subtyping** means conforming to a type (interface) signature, i.e. a set of APIs, and one can override part of the signature to achieve subtyping polymorphism.
2. **Subclassing** means implicit reuse of method implementations.

With the two benefits comes two different purposes for doing inheritance: subtyping oriented and code reuse oriented.

If code reuse is the *sole* purpose, subclassing may give one more than what he needs, i.e. some public methods of the parent class don't make much sense for the child class. In this case, instead of favoring composition over inheritance, composition is *demanded*. This is also where the "is-a" vs. "has-a" notion comes from.

So only when subtyping is purposed, i.e. to use the new class later in a polymorphic manner, do we face the problem of choosing inheritance or composition. This is the assumption that gets omitted in the shortened idiom under discussion.

To subtype is to conform to a type signature, this means composition has always to expose no less amount of APIs of the type. Now the trade offs kick in:

1. Inheritance provides straightforward code reuse if not overridden, while composition has to re-code every API, even if it's just a simple job of delegation.
2. Inheritance provides straightforward open recursion via the internal polymorphic site `this`, i.e. invoking overriding method (or even type) in another member function, either public or private (though discouraged). Open recursion can be simulated via composition, but it requires extra effort and may not always viable(?). This answer to a duplicated question talks something similar.

3. Inheritance exposes *protected* members. This breaks encapsulation of the parent class, and if used by subclass, another dependency between the child and its parent is introduced.

4. Composition has the befit of inversion of control, and its dependency can be injected dynamically, as is shown in decorator pattern and proxy pattern.

5. Composition has the benefit of combinator-oriented programming, i.e. working in a way like the composite pattern.

6. Composition immediately follows programming to an interface.

7. Composition has the benefit of easy multiple inheritance.

With the above trade offs in mind, we hence *prefer* composition over inheritance. Yet for tightly related classes, i.e. when implicit code reuse really make benefits, or the magic power of open recursion is desired, inheritance shall be the choice.

Share Improve this answer Follow

edited May 23, 2017 at 12:10

Community Bot
1 ● 1

answered Sep 14, 2015 at 5:22

lcn
2,279 ● 26 ● 41

1 I think you mean subclassing entails subtyping, as is the case with most OOP languages. To rephrase, "Subclassing means implicit reuse of method implementations **AND** conforming to a type (interface) signature. – wlnirvana Apr 18, 2022 at 6:17

Point number 3 does not break encapsulation as claimed, as it is the parent that decides whether members are protected or private. You want your parent class fully encapsulated? Declare everything private: job done. You want to use protected for the purpose for which it exists? Then use it, and rest assured that you aren't exposing anything that you didn't intend to be exposed. – Trevortni May 30, 2023 at 19:42

66

Inheritance is pretty enticing especially coming from procedural-land and it often looks deceptively elegant. I mean all I need to do is add this one bit of functionality to some other class, right? Well, one of the problems is that

inheritance is probably the worst form of coupling you can have

Your base class breaks encapsulation by exposing implementation details to subclasses in the form of protected members. This makes your system rigid and fragile. The more tragic flaw however is the new subclass brings with it all the baggage and opinion of the inheritance chain.

The article, Inheritance is Evil: The Epic Fail of the DataAnnotationsModelBinder, walks through an example of this in C#. It shows the use of inheritance when composition should have been used and how it could be refactored.

Share Improve this answer Follow

edited Mar 16, 2014 at 15:32

answered Jan 23, 2010 at 19:55

Mike Valenty

8,981 ● 2 ● 31 ● 32

5 Inheritance isn't good or bad, it's merely a special case of Composition. Where, indeed, the subclass is implementing a similar functionality to the superclass. If your proposed subclass is not re-implementing but merely *using* the functionality of the superclass, then you have used Inheritance incorrectly. That is the programmer's mistake, not a reflection on Inheritance. – iPherian Apr 19, 2017 at 22:31

This entire answer appears to be based on the author's ignorance of the difference between private and protected, as it gives literally no other problem than one issue (that could theoretically be) caused by not knowing basic keywords. – Trevortni May 30, 2023 at 19:52

64

When can you use composition?

You can always use composition. In some cases, inheritance is also possible and may lead to a more powerful and/or intuitive API, but composition is always an option.

When can you use inheritance?

It is often said that if "a bar is a foo", then the class `Bar` can inherit the class `Foo`. Unfortunately, this test alone is not reliable, use the following instead:

> 1. a bar is a foo, **AND**
> 2. bars can do everything that foos can do.

The first test ensures that all *getters* of `Foo` make sense in `Bar` (= shared properties), while the second test makes sure that all *setters* of `Foo` make sense in `Bar` (= shared functionality).

Example: Dog/Animal

A dog is an animal AND dogs can do everything that animals can do (such as breathing, moving, etc.).

Therefore, the class `Dog` **can** inherit the class `Animal`.

Counter-example: Circle/Ellipse

A circle is an ellipse BUT circles can't do everything that ellipses can do. For example, circles can't stretch, while ellipses can. Therefore, the class `Circle` **cannot** inherit the class `Ellipse`.

This is called the Circle-Ellipse problem, which isn't really a problem, but more an indication that "a bar is a foo" isn't a reliable test by itself. In particular, this example highlights that derived classes should *extend* the functionality of base classes, never *restrict* it. Otherwise, the base class couldn't be used polymorphically. Adding the test "bars can do everything that foos can do" ensures that polymorphic use is possible, and is equivalent to the Liskov Substitution Principle:

> Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it

When should you use inheritance?

Even if you *can* use inheritance doesn't mean you *should*: using composition is always an option. Inheritance is a powerful tool allowing implicit code reuse and dynamic dispatch, but it does come with a few disadvantages, which is why composition is often preferred. The trade-offs between inheritance and composition aren't obvious, and in my opinion are best explained in lcn's answer.

As a rule of thumb, I tend to choose inheritance over composition when polymorphic use is expected to be very common, in which case the power of dynamic dispatch can lead to a much more readable and elegant API. For example, having a polymorphic class `Widget` in GUI frameworks, or a polymorphic class `Node` in XML libraries allows to have an API which is much more readable and intuitive to use than what you would have with a solution purely based on composition.

Share Improve this answer Follow

edited Aug 7, 2020 at 9:53

answered Jun 3, 2016 at 22:23

Boris Dalstein
7,708 ● 4 ● 35 ● 60

The circle-ellipse and square-rectangle scenarios are poor examples. Subclasses are invariably more complex than their superclass, so the problem is contrived. This problem is solved by inverting the relationship. An ellipse derives from a circle and a rectangle derives from a square. It's extremely silly to use composition in these scenarios. – Fuzzy Logic Oct 18, 2018 at 1:12

---

2 @FuzzyLogic And if you are curious about what I would advocate for the specific case of Circle-Ellipse: I would advocate not implementing the Circle class. The issue with

inverting the relationship is that it also violates LSP: imagine the function `computeArea(Circle* c) { return pi * square(c->radius()); }`. It is obviously broken if passed an Ellipse (what does radius() even mean?). An Ellipse is not a Circle, and as such shouldn't derive from Circle. – Boris Dalstein Oct 18, 2018 at 9:59

3 @FuzzyLogic I disagree: you realize that this means that the class Circle anticipated the existence of the derived class Ellipse, and therefore provided `width()` and `height()`? What if now a library user decides to create another class called "EggShape"? Should it also derive from "Circle"? Of course not. An egg-shape is not a circle, and an ellipse is not a circle either, so none should derive from Circle since it breaks LSP. Methods performing operation on a Circle* class make strong assumptions about what a circle is, and breaking these assumptions will almost certainly lead to bugs. – Boris Dalstein Oct 18, 2018 at 16:54

2 You are awesome, loved this answer :) – Nom1fan Feb 1, 2022 at 13:39

1 @Nom1fan Thanks, I'm glad the answer was helpful! – Boris Dalstein Feb 2, 2022 at 15:35

50

In Java or C#, an object cannot change its type once it has been instantiated.

So, if your object need to appear as a different object or behave differently depending on an object state or conditions, then use **Composition**: Refer to State and Strategy Design Patterns.

If the object need to be of the same type, then use **Inheritance** or implement interfaces.

Share Improve this answer

Follow

answered Sep 8, 2008 at 2:25

dance2die
36.9k ● 39 ● 135 ● 197

10 +1 I've found less and less that inheritance works in most situations. I much prefer shared/inherited interfaces and composition of objects....or is it called aggregation? Don't ask me, I've got a EE degree!! – kenny May 21, 2009 at 10:46

I believe that this is the most common scenario where "composition over inheritance" applies since both could be fitting in theory. For instance, in a marketing system you might have a the concept of a `Client`. Then, a new concept of a `PreferredClient` pops up later on. Should `PreferredClient` inherit `Client`? A preferred client 'is a' client afterall, no? Well, not so fast... like you said objects cannot change their class at runtime. How would you model the `client.makePreferred()` operation? Perhaps the answer lies in using composition with a missing concept, an `Account` perhaps? – plalx Aug 31, 2015 at 17:23

Rather than having different type of `Client` classes, perhaps there's just one that encapsulates the concept of an `Account` which could be a `StandardAccount` or a `PreferredAccount`... – plalx Aug 31, 2015 at 17:25

39

Personally I learned to always prefer composition over inheritance. There is no programmatic problem you can solve with inheritance which you cannot solve with composition; though you may have to use Interfaces(Java) or Protocols(Obj-C) in some cases. Since C++ doesn't know any such thing, you'll have to use abstract base classes, which means you cannot get entirely rid of inheritance in C++.

Composition is often more logical, it provides better abstraction, better encapsulation, better code reuse (especially in very large projects) and is less likely to break anything at a distance just because you made an isolated change anywhere in your code. It also makes it easier to uphold the "*Single Responsibility Principle*", which is often summarized as "*There should never be more than one reason for a class to change.*", and it means that every class exists for a specific purpose and it should only have methods that are directly related to its purpose. Also having a very shallow inheritance tree makes it much easier to keep the overview even when your project starts to get really large. Many people think that inheritance represents our *real world* pretty well, but that isn't the truth. The real world uses much more composition than inheritance. Pretty much every real world object you can hold in your hand has been composed out of other, smaller real world objects.

There are downsides of composition, though. If you skip inheritance altogether and only focus on composition, you will notice that you often have to write a couple of extra code lines that weren't necessary if you had used inheritance. You are also sometimes forced to repeat yourself and this violates the *DRY Principle* (DRY = Don't Repeat Yourself). Also composition often requires delegation, and a method is just calling another method of another object with no other code surrounding this call.

Such "double method calls" (which may easily extend to triple or quadruple method calls and even farther than that) have much worse performance than inheritance, where you simply inherit a method of your parent. Calling an inherited method may be equally fast as calling a non-inherited one, or it may be slightly slower, but is usually still faster than two consecutive method calls.

You may have noticed that most OO languages don't allow multiple inheritance. While there are a couple of cases where multiple inheritance can really buy you something, but those are rather exceptions than the rule. Whenever you run into a situation where you think "multiple inheritance would be a really cool feature to solve this problem", you are usually at a point where you should re-think inheritance altogether, since even it may require a couple of extra code lines, a solution based on composition will usually turn out to be much more elegant, flexible and future proof.

Inheritance is really a cool feature, but I'm afraid it has been overused the last couple of years. People treated inheritance as the one hammer that can nail it all, regardless if it was actually a nail, a screw, or maybe a something completely different.

Share Improve this answer Follow

answered Jan 31, 2013 at 19:34

Mecki

133k ● 34 ● 259 ● 269

"Many people think that inheritance represents our real world pretty well, but that isn't the truth." So much this! Contrary to pretty much all the programming tutorials in the world ever, modeling real-world objects as inheritance chains is probably a bad idea in the long run. You should use inheritance only when there's an incredibly obvious, innate, simple is-a relationship. Like `TextFile` is a `File`. – neonblitzer May 6, 2020 at 10:22

@neonblitzer Even TextFile is-a File could be modeled without too much trouble via other techniques. For example, you could make "File" be an interface, with concrete TextFile and BinaryFile implementations, or, maybe a "File" is a class that can be instantiated with an instance of BinaryFileBehaviors or TextFileBehaviors (i.e. using the strategy pattern). I've given up on is-a, and now just follow the mantra of "inheritance is a last resort, use it when no other option will work sufficiently". – Scotty Jamison Apr 26, 2022 at 1:50

28

My general rule of thumb: *Before using inheritance, consider if composition makes more sense.*

Reason: *Subclassing usually means more complexity and connectedness, i.e. harder to change, maintain, and scale without making mistakes.*

A much more complete and concrete answer from Tim Boudreau of Sun:

> Common problems to the use of inheritance as I see it are:
>
> - *Innocent acts can have unexpected results* - The classic example of this is calls to

> overridable methods from the superclass constructor, before the subclasses instance fields have been initialized. In a perfect world, nobody would ever do that. This is not a perfect world.
>
> - *It offers perverse temptations for subclassers to make assumptions about order of method calls and such* - such assumptions tend not to be stable if the superclass may evolve over time. See also my toaster and coffee pot analogy.
> - *Classes get heavier* - you don't necessarily know what work your superclass is doing in its constructor, or how much memory it's going to use. So constructing some innocent would-be lightweight object can be far more expensive than you think, and this may change over time if the superclass evolves
> - *It encourages an explosion of subclasses*. Classloading costs time, more classes costs memory. This may be a non-issue until you're dealing with an app on the scale of NetBeans, but there, we had real issues with, for example, menus being slow because the first display of a menu triggered massive class loading. We fixed this by moving to more declarative syntax and other techniques, but that cost time to fix as well.

> - *It makes it harder to change things later* - if you've made a class public, swapping the superclass is going to break subclasses - it's a choice which, once you've made the code public, you're married to. So if you're not altering the real functionality to your superclass, you get much more freedom to change things later if you use, rather than extend the thing you need. Take, for example, subclassing JPanel - this is usually wrong; and if the subclass is public somewhere, you never get a chance to revisit that decision. If it's accessed as JComponent getThePanel() , you can still do it (hint: expose models for the components within as your API).
> - *Object hierarchies don't scale (or making them scale later is much harder than planning ahead)* - this is the classic "too many layers" problem. I'll go into this below, and how the AskTheOracle pattern can solve it (though it may offend OOP purists).
>
> ...
>
> My take on what to do, if you do allow for inheritance, which you may take with a grain of salt is:
>
> - Expose no fields, ever, except constants
> - Methods shall be either abstract or final

> - Call no methods from the superclass constructor
>
> ...
>
> all of this applies less to small projects than large ones, and less to private classes than public ones

Share Improve this answer Follow

answered May 21, 2012 at 1:59

Peter Tseng
14k ● 5 ● 71 ● 57

20

Inheritance is very powerful, but you can't force it (see: the circle-ellipse problem). If you really can't be completely sure of a true "is-a" subtype relationship, then it's best to go with composition.

Share Improve this answer Follow

answered Sep 8, 2008 at 2:29

yukondude
24.6k ● 13 ● 50 ● 58

19

If you want the canonical, textbook answer people have been giving since the rise of OOP (which you see many people giving in these answers), then apply the following rule: "if you have an is-a relationship, use inheritance. If you have a has-a relationship, use composition".

This is the traditional advice, and if that satisfies you, you can stop reading here and go on your merry way. For everyone else...

is-a/has-a comparisons have problems

For example:

- A square is-a rectangle, but if your rectangle class has `setWidth()`/`setHeight()` methods, then there's no reasonable way to make a `Square` inherit from `Rectangle` without breaking Liskov's substitution principle.
- An is-a relationship can often be rephrased to sound like a has-a relationship. For example, an employee is-a person, but a person also has-an employment status of "employed".
- is-a relationships can lead to nasty multiple inheritance hierarchies if you're not careful. After all, there's no rule in English that states that an object *is* exactly one thing.
- People are quick to pass this "rule" around, but has anyone ever tried to back it up, or explain why it's a good heuristic to follow? Sure, it fits nicely into the idea that OOP is supposed to model the real world, but that's not in-and-of-itself a reason to adopt a principle.

See this StackOverflow question for more reading on this subject.

To know when to use inheritance vs composition, we first need to understand the pros and cons of each.

The problems with implementation inheritance

Other answers have done a wonderful job at explaining the issues with inheritance, so I'll try to not delve into too many details here. But, here's a brief list:

- It can be difficult to follow a logic that weaves between base and sub-class methods.
- Carelessly implementing one method in your class by calling another overridable method will cause you to leak implementation details and break encapsulation, as the end-user could override your method and detect when you internally call it. (See "Effective Java" item 18).
- The fragile base problem, which simply states that your end-user's code will break if they happen to depend on the leakage of implementation details when you attempt to change them. To make matters worse, most OOP languages allow inheritance by default - API designers who aren't proactively preventing people from inheriting from their public classes need to be extra cautious whenever they

refactor their base classes. Unfortunately, the fragile base problem is often misunderstood, causing many to not understand what it takes to maintain a class that anyone can inherit from.

- The deadly diamond of death

The problems with composition

- It can sometimes be a little verbose.

That's it. I'm serious. This is still a real issue and can sometimes create conflict with the DRY principle, but it's generally not that bad, at least compared to the myriad of pitfalls associated with inheritance.

When should inheritance be used?

Next time you're drawing out your fancy UML diagrams for a project (if you do that), and you're thinking about adding in some inheritance, please adhere to the following advice: don't.

At least, not yet.

Inheritance is sold as a tool to achieve polymorphism, but bundled with it is this powerful code-reuse system, that frankly, most code doesn't need. The problem is, as soon as you publicly expose your inheritance hierarchy, you're locked into this particular style of code-reuse, even if it's overkill to solve your particular problem.

To avoid this, my two cents would be to never expose your base classes publicly.

- If you need polymorphism, use an interface.
- If you need to allow people to customize the behavior of your class, provide explicit hook-in points via the strategy pattern, it's a more readable way to accomplish this, plus, it's easier to keep this sort of API stable as you're in full control over what behaviors they can and can not change.
- If you're trying to follow the open-closed principle by using inheritance to avoid adding a much-needed update to a class, just don't. Update the class. Your codebase will be much cleaner if you actually take ownership of the code you're hired to maintain instead of trying to tack stuff onto the side of it. If you're scared about introducing bugs, then get the existing code under test.
- If you need to reuse code, start out by trying to use composition or helper functions.

Finally, if you've decided that there's no other good option, and you must use inheritance to achieve the code-reuse that you need, then you can use it, but, follow these four **P.A.I.L.** rules of restricted inheritance to keep it sane.

1. Use inheritance as a **private** implementation detail. Don't expose your base class publicly, use interfaces for that. This lets you freely add or remove inheritance as you see fit without making a breaking change.
2. Keep your base class **abstract**. It makes it easier to divide out the logic that needs to be shared from the logic that doesn't.
3. **Isolate** your base and child classes. Don't let your subclass override base class methods (use the strategy pattern for that), and avoid having them expect properties/methods to exist on each other, use other forms of code-sharing to achieve that. Use appropriate language features to force all methods on the base class to be non-overridable ("final" in Java, or non-virtual in C#).
4. Inheritance is a **last** resort.

The **Isolate** rule in particular may sound a little rough to follow, but if you discipline yourself, you'll get some pretty nice benefits. In particular, it gives you the freedom to avoid all of the main nasty pitfalls associated with the inheritance that were mentioned above.

- It's much easier to follow the code because it doesn't weave in and out of base/sub classes.

- You can not accidentally leak when your methods are internally calling other overridable methods if you never make any of your methods overridable. In other words, you won't accidentally break encapsulation.
- The fragile base class problem stems from the ability to depend on accidentally leaked implementation details. Since the base class is now isolated, it will be no more fragile than a class depending on another via composition.
- The deadly diamond of death isn't an issue anymore, since there's simply no need to have multiple layers of inheritance. If you have the abstract base classes B and C, which both share a lot of functionality, just move that functionality out of B and C and into a new abstract base class, class D. Anyone who inherited from B should update to inherit from both B and D, and anyone who inherited from C should inherit from C and D. Since your base classes are all private implementation details, it shouldn't be too difficult to figure out who's inheriting from what, to make these changes.

Conclusion

My primary suggestion would be to use your brain on this matter. What's far more important than a list of dos and don'ts about when to use inheritance is an intuitive understanding of inheritance and its associated pros and cons, along with a good understanding of the other tools out there that can be used instead of inheritance (composition isn't the only alternative. For example, the strategy pattern is an amazing tool that's forgotten far too often). Perhaps when you have a good, solid understanding of all of these tools, you'll choose to use inheritance more often than I would recommend, and that's completely fine. At least, you're making an informed decision, and aren't just using inheritance because that's the only way you know how to do it.

Further reading:

- An article I wrote on this subject, that dives even deeper and provides examples.
- A webpage talking about three different jobs that inheritance does, and how those jobs can be done via other means in the Go language.
- A list of reasons why it can be good to declare your class as non-inheritable (e.g. "final" in Java).
- The "Effective Java" book by Joshua Bloch, item 18, which discusses composition over inheritance, and some of the dangers of inheritance.

Share Improve this answer Follow

edited Sep 22, 2022 at 9:09
Dr.jacky
3,547 ● 6 ● 57 ● 96

answered Mar 22, 2022 at 7:17
Scotty Jamison
13.1k ● 2 ● 28 ● 31

18

Inheritance creates a strong relationship between a subclass and super class; subclass must be aware of super class'es implementation details. Creating the super class is much harder, when you have to think about how it can be extended. You have to document class invariants carefully, and state what other methods overridable methods use internally.

Inheritance is sometimes useful, if the hierarchy really represents a is-a-relationship. It relates to Open-Closed Principle, which states that classes should be closed for modification but open to extension. That way you can have polymorphism; to have a generic method that deals with super type and its methods, but via dynamic dispatch the method of subclass is invoked. This is flexible, and helps to create indirection, which is essential in software (to know less about implementation details).

Inheritance is easily overused, though, and creates additional complexity, with hard dependencies between classes. Also understanding what happens during execution of a program gets pretty hard due to layers and dynamic selection of method calls.

I would suggest using composing as the default. It is more modular, and gives the benefit of late binding (you can change the component dynamically). Also it's easier to test the things separately. And if you need to use a method from a class, you are not forced to be of certain form (Liskov Substitution Principle).

Share Improve this answer Follow

edited May 21, 2009 at 8:18

answered May 21, 2009 at 8:11

egaga
21.7k ● 10 ● 47 ● 61

4 It's worth noting that inheritance is not the only way to achieve polymorphism. The Decorator Pattern provides the appearance of polymorphism through composition. – BitMask777 Sep 20, 2012 at 20:16

1 @BitMask777: Subtype polymorphism is only one kind of polymorphism, another would be parametric polymorphism, you don't need inheritance for that. Also more importantly: when talking about inheritance, one means class inheritance; .i.e. you can have subtype polymorphism by having a common interface for multiple classes, and you don't get the problems of inheritance. – egaga Sep 23, 2012 at 11:41

2 @engaga: I interpreted your comment `Inheritance is sometimes useful... That way you can have polymorphism` as hard-linking the concepts of inheritance and polymorphism (subtyping assumed given the context). My comment was intended to point out what you clarify in your comment: that inheritance is not the only way to implement polymorphism, and in fact is not necessarily the

determining factor when deciding between composition and inheritance. – BitMask777 Oct 1, 2012 at 20:09

18

Suppose an aircraft has only two parts: an engine and wings.
Then there are two ways to design an aircraft class.

```
Class Aircraft extends Engine{
  var wings;
}
```

Now your aircraft can start with having fixed wings
and change them to rotary wings on the fly. It's essentially
an engine with wings. But what if I wanted to change
the engine on the fly as well?

Either the base class `Engine` exposes a mutator to change its
properties, or I redesign `Aircraft` as:

```
Class Aircraft {
  var wings;
  var engine;
}
```

Now, I can replace my engine on the fly as well.

Share Improve this answer Follow

edited May 6, 2011 at 17:33

Marc Mutz - mmutz
25.2k ● 12 ● 82 ● 94

answered Nov 29, 2010 at 10:54

simplfuzz
12.8k ● 25 ● 87 ● 140

Your post brings up a point I'd not considered before--to continue your an analogy of mechanical objects with multiple parts, on something like a firearm, there is generally one part marked with a serial number, whose serial number is considered to be that of the firearm as a whole (for a handgun, it would typically be the frame). One may replace all the other parts and still have the same firearm, but if the frame cracks and needs to be replaced, the result of assembling a new frame with all the other parts from the original gun would be a new gun. Note that... – supercat Aug 19, 2012 at 16:56

...the fact that multiple parts of a gun might have serial numbers marked on them does not mean that a gun can have multiple identities. Only the serial number on the frame identifies the gun; the serial number on any other part identifies which gun those parts were manufactured to be assembled with, which may not be the gun to which they are assembled at any given time. – supercat Aug 19, 2012 at 17:00

1 In this particular case of aircraft I would definitely not recommend to 'change the engine on the fly' – Paramvir Singh Karwal Aug 30, 2020 at 16:00

13

You need to have a look at **The Liskov Substitution Principle** in Uncle Bob's SOLID principles of class design. :)

edited Dec 23, 2012 at 22:35

Share Improve this answer

Follow

Rob W
349k ● 87 ● 807 ● 682

answered Oct 5, 2010 at 11:57

nabeelfarid
4,224 ● 6 ● 43 ● 62

10

To address this question from a different perspective for newer programmers:

Inheritance is often taught early when we learn object-oriented programming, so it's seen as an easy solution to a common problem.

> I have three classes that all need some common functionality. So if I write a base class and have them all inherit from it, then they will all have that functionality and I'll only need to maintain it in once place.

It sounds great, but in practice it almost never, ever works, for one of several reasons:

- We discover that there are some other functions that we want our classes to have. If the way that we add functionality to classes is through inheritance, we have to decide - do we add it to the existing base class, even though not every class that inherits from it needs that functionality? Do we create another base class? But what about classes that already inherit from the other base class?

- We discover that for just one of the classes that inherits from our base class we want the base class to behave a little differently. So now we go back and tinker with our base class, maybe adding some virtual methods, or even worse, some code that says, "If I'm inherited type A, do this, but if I'm inherited type B, do that." That's bad for lots of reasons. One is that every time we change the base class, we're effectively changing every inherited class. So we're really changing class A, B, C, and D because we need a slightly different behavior in class A. As careful as we think we are, we might break one of those classes for reasons that have nothing to do with those classes.
- We might know why we decided to make all of these classes inherit from each other, but it might not (probably won't) make sense to someone else who has to maintain our code. We might force them into a difficult choice - do I do something really ugly and messy to make the change I need (see the previous bullet point) or do I just rewrite a bunch of this.

In the end, we tie our code in some difficult knots and get no benefit whatsoever from it except that we get to say, "Cool, I learned about inheritance and now I used it." That's not meant to be condescending because we've all done it. But we all did it because no one told us not to.

As soon as someone explained "favor composition over inheritance" to me, I thought back over every time I tried to share functionality between classes using inheritance and realized that most of the time it didn't really work well.

The antidote is the Single Responsibility Principle. Think of it as a constraint. My class *must* do one thing. I *must* be able to give my class a name that somehow describes that one thing it does. (There are exceptions to everything, but absolute rules are sometimes better when we're learning.) It follows that I cannot write a base class called `ObjectBaseThatContainsVariousFunctionsNeededByDifferentClasses`. Whatever distinct functionality I need must be in its own class, and then other classes that need that functionality can depend on that class, *not* inherit from it.

At the risk of oversimplifying, that's composition - composing multiple classes to work together. And once we form that habit we find that it's much more flexible, maintainable, and testable than using inheritance.

Share Improve this answer Follow

answered May 13, 2016 at 20:24

Scott Hannen
29.1k ● 4 ● 51 ● 68

That classes can't use multiple base classes is not a poor reflection on Inheritance but rather a poor reflection on a particular language's lack of capability. – iPherian Apr 19, 2017 at 22:22

In the time since writing this answer, I read this post from "Uncle Bob" which addresses that lack of capability. I've never used a language that allows multiple inheritance. But looking back, the question is tagged "language agnostic" and my answer assumes C#. I need to broaden my horizons. – Scott Hannen Apr 23, 2017 at 14:04

The second bullet point merely seems to be explaining the need for overrides, which I've never heard of a language that doesn't provide. – Trevortni May 30, 2023 at 19:26

That works if it was understood in advance which behaviors we'll need to override and the class is organized to make that simple. Otherwise the result is confusing control flow or we still end up modifying other classes. It's possible to do it for the right reasons and get it right. More often we couple unrelated classes to each other via a base class and it then the problems start to show up. – Scott Hannen May 30, 2023 at 22:57

8

When you want to "copy"/Expose the base class' API, you use inheritance. When you only want to "copy" functionality, use delegation.

One example of this: You want to create a Stack out of a List. Stack only has pop, push and peek. You shouldn't use inheritance given that you don't want push_back, push_front, removeAt, et al.-kind of functionality in a Stack.

Share Improve this answer Follow

answered May 7, 2009 at 1:43

Anzurio
17k ● 3 ● 40 ● 49

@Anzurio....How do you differentiate API vs functionlaity? According to me exposed methods of a class we call them as api methods of that class. These are the functionality of the class as well. If you use composition, You use public methods of rhe class, we have already called them API. – sdindiver Jun 26, 2020 at 18:06

1 @sdindiver sorry for the confusion. The point I was trying to make is that, when using inheritance, you will expose the complete API of the parent class but, if the child class needn't expose that full parent's API, you use composition instead such that the child class will have access to the parent class' functionality without exposing its full API. – Anzurio Jun 27, 2020 at 1:11

7

These two ways can live together just fine and actually support each other.

Composition is just playing it modular: you create interface similar to the parent class, create new object and delegate calls to it. If these objects need not to know of each other, it's quite safe and easy to use composition. There are so many possibilites here.

However, if the parent class for some reason needs to access functions provided by the "child class" for inexperienced programmer it may look like it's a great place to use inheritance. The parent class can just call it's own abstract "foo()" which is overwritten by the subclass and then it can give the value to the abstract base.

It looks like a nice idea, but in many cases it's better just give the class an object which implements the foo() (or even set the value provided the foo() manually) than to inherit the new class from some base class which requires the function foo() to be specified.

Why?

Because inheritance is a poor way of moving information.

The composition has a real edge here: the relationship can be reversed: the "parent class" or "abstract worker" can aggregate any specific "child" objects implementing certain interface + **any child can be set inside any other type of parent, which accepts it's type**. And there can be any number of objects, for example MergeSort or QuickSort could sort any list of objects implementing an abstract Compare -interface. Or to put it another way: any group of objects which implement "foo()" and other group of objects which can make use of objects having "foo()" can play together.

I can think of three real reasons for using inheritance:

1. You have many classes with **same interface** and you want to save time writing them
2. You have to use same Base Class for each object
3. You need to modify the private variables, which can not be public in any case

If these are true, then it is probably necessary to use inheritance.

There is nothing bad in using reason 1, it is very good thing to have a solid interface on your objects. This can be done using composition or with inheritance, no problem - if this interface is simple and does not change. Usually inheritance is quite effective here.

If the reason is number 2 it gets a bit tricky. Do you really only need to use the same base class? In general, just using the same base class is not good enough, but it may be a requirement of your framework, a design consideration which can not be avoided.

However, if you want to use the private variables, the case 3, then you may be in trouble. **If you consider global variables unsafe, then you should consider using inheritance to get access to private variables also unsafe**. Mind you, global variables are not all THAT bad - databases are essentially big set of global variables. But if you can handle it, then it's quite fine.

Share Improve this answer Follow

answered Apr 16, 2013 at 20:15

Tero Tolonen
4,254 ● 4 ● 28 ● 33

6

Aside from is a/has a considerations, one must also consider the "depth" of inheritance your object has to go through. Anything beyond five or six levels of inheritance deep might cause unexpected casting and boxing/unboxing problems, and in those cases it might be wise to compose your object instead.

Share Improve this answer

Follow

answered Sep 8, 2008 at 3:00

Jon Limjap
95.3k ● 15 ● 103 ● 153

6

When you have an **is-a** relation between two classes (example dog is a canine), you go for inheritance.

On the other hand when you have **has-a** or some adjective relationship between two classes (student has courses) or (teacher studies courses), you chose composition.

Share Improve this answer

Follow

edited Dec 15, 2017 at 14:06

Ravindra babu
38.9k ● 11 ● 256 ● 219

answered Dec 30, 2013 at 1:57

Amir Aslam
383 ● 4 ● 9

You said inheritance and inheritance. don't you mean inheritance and composition? – trevorKirkby Dec 30, 2013 at 2:21

No, you don't. You can just as well define a canine interface and let every dog implement it and you'll end up with more SOLID code. – markus Oct 4, 2015 at 13:49

5

A simple way to make sense of this would be that inheritance should be used when you need an object of your class to have the same *interface* as its parent class, so that it can thereby be treated as an object of the parent class (upcasting). Moreover, function calls on a derived class object would remain the same everywhere in code, but the specific method to call would be determined at runtime (i.e. the low-level *implementation* differs, the high-level *interface* remains the same).

Composition should be used when you do not need the new class to have the same interface, i.e. you wish to conceal certain aspects of the class' implementation which the user of that class need not know about. So composition is more in the way of supporting *encapsulation* (i.e. concealing the implementation) while inheritance is meant to support *abstraction* (i.e. providing a simplified representation of something, in this case the **same** interface for a range of types with different internals).

Share Improve this answer Follow

edited Mar 20, 2015 at 13:57

answered Jun 4, 2014 at 11:36

Yash Sampat
30.6k ● 12 ● 98 ● 121

+1 for mention of interface. I use this approach often to hide existing classes and make my new class properly unit testable by mocking out the object used for composition. This requires the owner of the new object to pass it the candidate parent class instead. – Kell Jun 13, 2014 at 10:39

4

Subtyping is appropriate and more powerful where the invariants can be enumerated, else use function composition for extensibility.

Share Improve this answer Follow

edited May 23, 2017 at 12:34
Community Bot
1 ● 1

answered Dec 2, 2011 at 7:40
Shelby Moore III
6,121 ● 1 ● 34 ● 36

4

I agree with @Pavel, when he says, there are places for composition and there are places for inheritance.

I think inheritance should be used if your answer is an affirmative to any of these questions.

- Is your class part of a structure that benefits from polymorphism ? For example, if you had a Shape class, which declares a method called draw(), then we clearly need Circle and Square classes to be subclasses of Shape, so that their client classes would depend on Shape and not on specific subclasses.
- Does your class need to re-use any high level interactions defined in another class ? The template method design pattern would be impossible to implement without inheritance. I believe all extensible frameworks use this pattern.

However, if your intention is purely that of code re-use, then composition most likely is a better design choice.

Share Improve this answer Follow

answered Dec 7, 2011 at 10:40

Parag
12.4k ● 16 ● 60 ● 76

4

Inheritance is a very powerfull machanism for code reuse. But needs to be used properly. I would say that inheritance is used correctly if the subclass is also a subtype of the parent class. As mentioned above, the Liskov Substitution Principle is the key point here.

Subclass is not the same as subtype. You might create subclasses that are not subtypes (and this is when you should use composition). To understand what a subtype is, lets start giving an explanation of what a type is.

When we say that the number 5 is of type integer, we are stating that 5 belongs to a set of possible values (as an example, see the possible values for the Java primitive types). We are also stating that there is a valid set of methods I can perform on the value like addition and subtraction. And finally we are stating that there are a set of properties that are always satisfied, for example, if I add the values 3 and 5, I will get 8 as a result.

To give another example, think about the abstract data types, Set of integers and List of integers, the values they can hold are restricted to integers. They both support a set of methods, like add(newValue) and size(). And they both have different properties (class invariant), Sets does not allow duplicates while List does allow duplicates (of course there are other properties that they both satisfy).

Subtype is also a type, which has a relation to another type, called parent type (or supertype). The subtype must satisfy the features (values, methods and properties) of the parent type. The relation means that in any context where the supertype is expected, it can be substitutable by a subtype, without affecting the behaviour of the execution. Let’s go to see some code to exemplify what I’m saying. Suppose I write a List of integers (in some sort of pseudo language):

```
class List {
  data = new Array();

  Integer size() {
    return data.length;
  }

  add(Integer anInteger) {
    data[data.length] = anInteger;
  }
}
```

Then, I write the Set of integers as a subclass of the List of integers:

```
class Set, inheriting from: List {
  add(Integer anInteger) {
     if (data.notContains(anInteger)) {
       super.add(anInteger);
     }
```

```
    }
}
```

Our Set of integers class is a subclass of List of Integers, but is not a subtype, due to it is not satisfying all the features of the List class. The values, and the signature of the methods are satisfied but the properties are not. The behaviour of the add(Integer) method has been clearly changed, not preserving the properties of the parent type. Think from the point of view of the client of your classes. They might receive a Set of integers where a List of integers is expected. The client might want to add a value and get that value added to the List even if that value already exist in the List. But her wont get that behaviour if the value exists. A big suprise for her!

This is a classic example of an improper use of inheritance. Use composition in this case.

(a fragment from: use inheritance properly).

Share Improve this answer Follow

answered Aug 31, 2013 at 13:41

Enrique Molinari
307 ● 3 ● 8

4

Even though Composition is preferred, I would like to highlight pros of *Inheritance* and cons of *Composition*.

Pros of Inheritance:

1. It establishes a logical "**IS A"** relation. If *Car* and *Truck* are two types of *Vehicle* (base class), child class **IS A** base class.

   i.e.

   Car is a Vehicle

   Truck is a Vehicle

2. With inheritance, you can define/modify/extend a capability

   1. Base class provides no implementation and sub-class has to override complete method (abstract) => ***You can implement a contract***

   2. Base class provides default implementation and sub-class can change the behaviour => ***You can re-define contract***

   3. Sub-class adds extension to base class implementation by calling super.methodName() as first statement => ***You can extend a contract***

   4. Base class defines structure of the algorithm and sub-class will override a part of algorithm => ***You can implement Template_method without change in base class skeleton***

Cons of Composition:

1. In inheritance, subclass can directly invoke base class method even though it's not implementing base class method because of ***IS A*** relation. If you use

composition, you have to add methods in container class to expose contained class API

e.g. If *Car* contains *Vehicle* and if you have to get price of the *Car*, which has been defined in *Vehicle*, your code will be like this

```
class Vehicle{
     protected double getPrice(){
          // return price
     }
}

class Car{
     Vehicle vehicle;
     protected double getPrice(){
          return vehicle.getPrice();
     }
}
```

Share Improve this answer Follow

edited Dec 13, 2016 at 6:22

answered Sep 22, 2016 at 9:31

Ravindra babu
38.9k ● 11 ● 256 ● 219

I think it doesn't answer the question – almanegra Mar 15, 2017 at 14:03

You can have re-look at OP question. I have addressed : What trade-offs are there for each approach?
– Ravindra babu Mar 15, 2017 at 14:34

1 As you mentioned, you talk only about "pros of Inheritance and cons of Composition", and not the tradeoffs for EACH

approach or the cases where you should use one over another – almanegra Mar 15, 2017 at 15:08

pros and cons provides trade-off since pros of inheritance is cons of composition and cons of composition is pros of inheritance. – Ravindra babu Mar 15, 2017 at 15:17

i am late to the party but; since cons of composition is pros of inheritence, you didnt talk about cons of inheritance. you only talked about pros of inheritance, twice. – numan Feb 3, 2022 at 13:03

3

A rule of thumb I have heard is inheritance should be used when its a "is-a" relationship and composition when its a "has-a". Even with that I feel that you should always lean towards composition because it eliminates a lot of complexity.

Share Improve this answer Follow

answered Sep 8, 2008 at 2:14

Evrhet Milam

2

As many people told, I will first start with the check - whether there exists an "is-a" relationship. If it exists I usually check the following:

> Whether the base class can be instantiated. That is, whether the base class can be non-abstract. If it can be non-abstract I usually prefer composition

E.g 1. Accountant **is an** Employee. But I will **not** use inheritance because a Employee object can be instantiated.

E.g 2. Book **is a** SellingItem. A SellingItem cannot be instantiated - it is abstract concept. Hence I will use inheritacne. The SellingItem is an **abstract base class (or interface** in C#)

What do you think about this approach?

Also, I support @anon answer in [Why use inheritance at all?]()

> The main reason for using inheritance is not as a form of composition - it is so you can get polymorphic behaviour. If you don't need polymorphism, you probably should not be using inheritance.

@MatthieuM. says in https://softwareengineering.stackexchange.com/questions/12439/code-smell-inheritance-abuse/12448#comment303759_12448

> The issue with inheritance is that it can be used for two orthogonal purposes:
>
> interface (for polymorphism)
>
> implementation (for code reuse)

REFERENCE

1. Which class design is better?

2. Inheritance vs. Aggregation

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 ● 1

answered Aug 1, 2012 at 11:15
LCJ
22.7k ● 69 ● 266 ● 428

1 I'm not sure why 'the base class is abstract?' figures into the discussion.. the LSP: will all functions that operate on Dogs work if Poodle objects are passed in ? If yes, then Poodle is can be substituted for Dog and hence can inherit from Dog. – Gishu Aug 1, 2012 at 12:50

@Gishu Thanks. I will definitely look into LSP. But before that, can you please provide an "example where inheritance is proper in which base class cannot be abstract". What I think is, inheritance is applicable only if base class is abstract. If base class need to be instantiated separately, do not go for inheritance. That is, even though Accountant is an Employee, do not use inheritance. – LCJ Aug 2, 2012 at 7:11

1 been reading WCF lately. An example in the .net framework is SynchronizationContext (base + can be instantiated) which queues work onto a ThreadPool thread. Derivations include WinFormsSyncContext (queue onto UI Thread) and DispatcherSyncContext (queue onto WPF Dispatcher) – Gishu Aug 2, 2012 at 9:05

@Gishu Thanks. However, it would be more helpful if you can provide a scenario based on Bank domain, HR domain, Retail domain or any other popular domain. – LCJ Aug 2, 2012 at 9:08

1 Sorry. I'm not familiar with those domains.. Another example if the previous one was too obtuse is the Control class in Winforms/WPF. The base/generic control can be instantiated. Derivations include Listboxes, Textboxes, etc. Now that I think of it, the Decorator Design pattern is a good example IMHO and useful too. The decorator derives from the non-abstract object that it wants to wrap/decorate. – Gishu Aug 2, 2012 at 9:28

2

Composition v/s Inheritance is a wide subject. There is no real answer for what is better as I think it all depends on the design of the system.

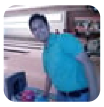
Generally type of relationship between object provide better information to choose one of them.

If relation type is "IS-A" relation then Inheritance is better approach. otherwise relation type is "HAS-A" relation then composition will better approach.

Its totally depend on entity relationship.

Share Improve this answer Follow

answered Mar 18, 2014 at 9:39



Shami Qureshi
69 ● 5