Composition over Inheritance

<http://programmers.stackexchange.com/questions/134097/why-should-i-prefer-composition-over-inheritance>

But, the reasons to prefer composition (in many circumstances) are profound.

When you implement an interface or extend a class you are generally declaring your intention to create a subtype. True subtypes are governed by *Liskov's Substitution Principle*, which says that if you can prove something about all objects in a supertype you can prove it about all instances in a subtype.

**Inheritance is a dangerous** like all very powerful things inheritance has the power to cause havoc. For example, suppose you override a method when inheriting from some class: all is well and good until some other method of that class assumes the method you inherit to behave a certain way, after all that is how the author of the original class designed it. You can partially protect against this by declaring all methods called by another of your methods private or non-virtual (final), unless they are*designed* to be overridden. Even this though isn't always good enough. Sometimes you might see something like this (in pseudo Java, hopefully readable to C++ and C# users)

interface UsefulThingsInterface {

void doThings();

void doMoreThings();

}

...

class WayOfDoingUsefulThings implements UsefulThingsInterface{

private foo stuff;

public final int getStuff();

void doThings(){

//modifies stuff, such that ...

...

}

...

void doMoreThings(){

//ignores stuff

...

}

}

you think this is lovely, and have your own way of doing "things", but you use inheritance to acquire the ability to do "moreThings",

class MyUsefulThings extends WayOfDoingUsefulThings{

void doThings {

//my way

}

}

And all is well and good. WayOfDoingUsefulThings was designed in such a way replacing one method doesn't change the semantics of any other...except wait, no it wasn't. It just looks like it was, but doThings changed mutable state that mattered so, even though it didn't call any override-able functions,

void dealWithStuff(WayOfDoingUsefulThings bar){

bar.doThings()

use(bar.getStuff());

}

now does something different than expected when you pass it a MyUsefulThings. Whats worse, you might not even know that WayOfDoingUsefulThings made those promises. Maybe dealWithStuffcomes from the same library as WayOfDoingUsefulThings and getStuff() isn't even exported by the library (think of *friend classes* in C++). Worse still, you have defeated the static checks of the language without realizing it: dealWithStuff took a WayOfDoingUsefulThings just to make sure that it would have a getStuff() function that behaved a certain way.

Using composition

class MyUsefulThings implements UsefulThingsInterface{

private way = new WayOfDoingUsefulThings()

void doThings() {

//my way

}

void doMoreThings() {

this.way.doMoreThings();

}

}

brings back static type safety. In general composition is easier to use and safer than inheritance when implementing subtyping. It also lets you override final methods which means that you should feel free to declare *everything* final/non-virtual except in interfaces the vast majority of the time.

In a better world languages would automatically insert the boilerplate with a delegation keyword. Most don't, so a downside is bigger classes. Although, you can get your IDE to write the delegating instance for you.

Now, life isn't just about polymorphism. You can don't need to subtype all the time. The goal of polymorphism is generally **code reuse** but it isn't the only way to achieve that goal. Often time, it makes sense to use composition, without subtype polymorphism, as a way of managing functionality.

Also, behavioral inheritance does have its uses. It is one of the most powerful ideas in computer science. Its just that, most of the time, good OOP applications can be written using only using interface inheritance and compositions. The two principles

1. Ban inheritance or design for it
2. Prefer composition

are a good guide for the reasons above, and don't incur any substantial costs.

**Delegation is one example of a way to use composition instead of inheritance. Delegation lets you modify the behavior of a class without subclassing**. Consider a class that provides a network connection, NetStream. It might be natural to subclass NetStream to implement a common network protocol, so you might come up with FTPStream and HTTPStream. But instead of creating a very specific HTTPStream subclass for a single purpose, say, UpdateMyWebServiceHTTPStream, it's often better to use a plain old instance of HTTPStream along with a delegate that knows what to do with the data it receives from that object. One reason it's better is that it avoids a proliferation of classes that have to be maintained but which you'll never be able to reuse. Another reason is that the object that serves as the delegate can also be responsible for other things, such as managing the data received from the web service.

<http://programmers.stackexchange.com/questions/65179/where-does-this-concept-of-favor-composition-over-inheritance-come-from/65183#65183>

"Composition over inheritance" is a short (and apparently misleading) way of saying "When feeling that the data (or behaviour) of a class should be incorporated into another class, always consider using composition before blindly applying inheritance".

Why is this true ? **Because inheritance creates tight, compile-time coupling between the 2 classes**. **Composition in contrast is loose coupling, wich among others enables clear separation of concerns, the possibility of switching dependencies at runtime and easier, more isolated dependency testability.**

A (metaphorical) example could be :

"I have a Snake class and I want to include as part of that class what happens when the Snake bites. I would be tempted to have the Snake inherit a BiterAnimal class that has the Bite() method and override that method to reflect venomous bite. But Composition over Inheritance warns me that I should try to use composition instead... In my case, this could translate into the Snake having a Bite member. Bite class could be abstract (or an interface) with several subclasses. This would allow me nice things like having VenomousBite and DryBite subclasses and being able to change bite on the same Snake instance as the snake grows of age. Plus handling all the effects of a Bite in its own separate class could allow me to reuse it in that Frost class, because frost bites but isn't a BiterAnimal, and so on..."

Some possible arguments for composition:

**Composition is slightly more language / framework agnostic**  
Inheritance and what it enforces / requires / enables will differ between languages in terms of what the sub/superclass have access to and what performance implications it may have wrt virtual methods etc. Composition is quite basic and requires very little language support, and thus implementations across different platforms / frameworks can share composition patterns more easily.

**Composition is a very simple and tactile way of building objects**  
Inheritance is relatively easy to understand, but still not as easily demonstrated in real life. Many objects in real life can be broken down into parts and composed. Say a bicycle can be built using two wheels, a frame, a seat, a chain etc. Easily explained by composition. Whereas in an inheritance metaphor you could say that a bicycle extends a unicycle, somewhat feasible but still much further from the real picture than composition (obviously this is not a very good inheritance example, but the point remains the same). Even the word inheritance (at least of most US English speakers I would expect) automatically invokes a meaning along the lines "Something passed down from a deceased relative" which has some correlation with its meaning in software, but still only loosely fits.

**Composition is almost always more flexible**  
Using composition you can always choose to define your own behavior or simply expose that part of your composed parts. This way you face none of the restrictions that may be imposed by an inheritance hierarchy (virtual vs. non-virtual etc.)

So, it could be because Composition is naturally a simpler metaphor that has less theoretical constraints than inheritance. Furthermore, these particular reasons may be more apparent during design time, or possibly stick out when dealing with some of the pain points of inheritance.

*Disclaimer:*  
Obviously its not this clear cut / one way street. Each design merits evaluation of several patterns / tools. Inheritance is widely used, has lots of benefits and many times is more elegant than composition. These are just some possible reasons one could use when favoring composition.

Perhaps you just noticed people saying this in the last few months, but it has been known to good programmers for a lot longer than that. I've certainly been saying it where appropriate for about a decade.

The point of the concept is that there is a large conceptual overhead to inheritance. When you are using inheritance, then every single method call has an implicit dispatch in it. If you have deep inheritance trees, or multiple dispatch, or (even worse) both, then figuring out where the particular method will dispatch to in any particular call can become a royal PITA. It makes correct reasoning about the code more complex, and it makes debugging harder.

Let me give a simple example to illustrate. Suppose that deep in an inheritance tree, someone named a method foo. Then someone else comes along and adds foo at the top of the tree, but doing something different. (This case is more common with multiple inheritance.) Now that person working at the root class has broken the obscure child class and probably doesn't realize it. You could have 100% coverage with unit tests and not notice this breakage because the person at the top wouldn't think of testing the child class, and the tests for the child class don't think of testing the new methods created at the top. (Admittedly there are ways to write unit tests that will catch this, but there are also cases where you can't easily write tests that way.)

By contrast when you use composition, at each call it is usually clearer what you are dispatching the call to. (OK, if you're using inversion of control, for instance with dependency injection, then figuring out where the call goes can also get problematic. But usually it is simpler to figure out.) This makes reasoning about it easier. As a bonus, composition results in having methods segregated from each other. The above example should not happen there because the child class would move off to some obscure component, and there is never a question about whether the call to foo was intended for the obscure component or the main object.

Now you are absolutely right that inheritance and composition are two very different tools that serve two different types of things. Sure inheritance carries conceptual overhead, but when it is the right tool for the job, it carries less conceptual overhead than trying to not use it and do by hand what it does for you. Nobody who knows what they are doing would say that you should never use inheritance. But be sure it is the right thing to do.

Unfortunately many developers learn about object oriented software, learn about inheritance, and then go out to use their new axe as often as possible. Which means that they try to use inheritance where composition was the right tool. Hopefully they will learn better in time, but frequently this does not happen until after a few removed limbs, etc. Telling them up front that it is a bad idea tends to speed up the learning process and reduce injuries.

<http://stackoverflow.com/questions/49002/prefer-composition-over-inheritance>

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 Class A.  
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 only 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](http://en.wikipedia.org/wiki/Liskov_substitution_principle) as a test for 'Should I be inheriting from this type?'

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.

**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's 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.

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](http://www.javalobby.org/forums/thread.jspa?forumID=61&threadID=16487#91822172) 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](http://www.javalobby.org/forums/thread.jspa?threadID=16036&messageID=91819530#91819530).
* 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

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.

**Favor Composition Over Inheritance**

<http://codingdelight.com/2014/01/16/favor-composition-over-inheritance-part-1/>

“Favor composition over inheritance” is a phrase that I hear spoken a lot but which describes a concept I rarely actually see in real world code.  Every developer seems to know about it but few developers seem to actually put it into practice. This post will be looking at inheritance and some of the pitfalls of trying to create your domain model primarily through inheritance.

I can only assume that the reason why inheritance is so overused in real world code is due to the way that it is taught.  Back, far too many years ago, while I was still studying at university, the concepts of inheritance and polymorphism where both taught side by side, very early in the object oriented programming course.  It seems as though these lessons were particularly memorable, because so much real-world code has giant inheritance chains.  We have ObscuredItems inheriting from DataItems inheriting from BasicItems which inherit from Items which inherit from BaseObjects.  Often times you will have to go five or six classes deep to find the root cause of a bug.

Favoring composition over inheritance helps us flatten those structures.  To illustrate this, I am going to take a look at a very simple problem.

I want to model real-world cars.  Specifically, I initially want to model the changes that a driver is applying to each car.  The customer in this case only wants me to model a Toyota Corolla.  The class should also have a string for the manufacturer name.  In this case I will deliberately favor inheritance.

Firstly, I’ll set up an abstract BaseCar class where I will define what I need: 4 wheels, a method for acceleration, a method for turning left and a method for turning right.

abstract class BaseCar

{

public Wheel FrontLeft { get; protected set; }

public Wheel FrontRight { get; protected set; }

public Wheel RearLeft { get; protected set; }

public Wheel RearRight { get; protected set; }

public abstract string Manufacturer { get; }

public abstract void TurnLeft();

public abstract void TurnRight();

public abstract void Accelerate(double kmsPerHour);

}

Next up, I will implement the functionality in my concrete car class.

abstract class Car : BaseCar

{

protected Car()

{

this.FrontLeft = new Wheel();

this.FrontRight = new Wheel();

this.BackLeft = new Wheel();

this.BackRight = new Wheel();

}

public override void TurnLeft(double degrees)

{

this.FrontLeft.TurnLeft(degrees);

this.FrontRight.TurnLeft(degrees);

}

public override void TurnRight(double degrees)

{

this.FrontRight.TurnRight(degrees);

this.FrontLeft.TurnRight(degrees);

}

public override void Accelerate(double kph)

{

this.FrontLeft.Rotate(kph);

this.FrontRight.Rotate(kph);

}

}

Finally, I will then implement a concrete ToyotaCorolla class.

class ToyotaCorolla : Car

{

public override string Manufacturer

{

get { return "Toyota"; }

}

}

All fairly straight forward.  We have a nice little inheritance chain that seems to make sense (although perhaps I could have done away with the BaseCar class).

I submit this code to the customer and they are very happy.  They’re so happy that they will pay me more money to add extra features.  They now want me to implement a ToyotaCorollaSports class, which contains a rear-wheel drive version of the car.  In all other aspects it’s the same car, it’s only that the rear wheels are now powering the car.

Simple enough, I can inherit from the ToyotaCorolla and override the accelerate method.  I make the change and submit it to the customer.  They like what they see.

class ToyotaCorollaSports : ToyotaCorolla

{

public override void Accelerate(double kph)

{

this.RearLeft.Rotate(kph);

this.RearRight.Rotate(kph);

}

}

Two weeks later the customer has come back.  They love what we’ve done and want us to implement HondaCivic and HondaCivicSports classes.  Now we run into our first real problem.  The rear-wheel drive code is actually in the concrete ToyotaCorollaSports class.  What we need to do now is actually refactor our Car class into FrontWheelDriveCar and RearWheelDriveCar.  HondaCivic and ToyotaCorolla can then both inherit from the FrontWheelDriveCar class and the sports editions can both inherit from the RearWheelDriveCar class.

abstract class FrontWheelDriveCar : Car

{

public override void Accelerate(double kph)

{

this.FrontLeft.Rotate(kph);

this.FrontRight.Rotate(kph);

}

}

abstract class RearWheelDriveCar : Car

{

public override void Accelerate(double kph)

{

this.RearLeft.Rotate(kph);

this.RearRight.Rotate(kph);

}

}

Our customers are very happy with what we have done so far and decide to commission us to create a four wheel drive Mitsubishi Titan.

So we now have to create an AllWheelDriveCar and inherit from that for our Mitsubishi Titan.

abstract class AllWheelDriveCar : Car

{

public override void Accelerate(double kph)

{

this.FrontLeft.Rotate(kph);

this.FrontRight.Rotate(kph);

this.RearLeft.Rotate(kph);

this.RearRight.Rotate(kph);

}

}

class MitsubishiTitan : AllWheelDriveCar

{

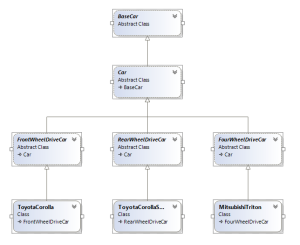
public override string Manufacturer

{

get { return "Mitsubishi"; }

}

}

[](http://codingdelightdotcom.files.wordpress.com/2014/01/class-diagram.png)  
Finally, the customer comes along and asks us to model for them a new experimental four wheel drive buggy that is designed to go on sand and turns with all four wheels.  A two wheel drive buggy is also available (it also turns with all four wheels).

Now we have a problem.  We will obviously be deriving from our four wheel drive class for the four wheel drive model and the two wheel drive class for the two wheel drive model, but we will also have to override our TurnLeft and TurnRight methods in both classes, and the code will be duplicated.

Arggg!  Duplicated code is the enemy of maintainability. The best solution to this problem is to switch our thinking. Instead of inheriting all of our functionality, we should compose our classes from pieces of related functionality.

Next up: a better way to solve the same problem.

**Composition over Inheritance**

In today’s article, I would like to talk about something which I see that, sometimes, developers aren’t too familiar with. Most of us know what the traditional concept of **inheritance** is: a child class extends a parent class. And, obviously, multiple children classes can extend the same parent class.

What some developers don’t seem to be always aware of (or, perhaps, they just aren’t entirely sure how to use it), is the idea of **composition**.

## What is Composition

Composition can be explained by a child class “incorporating” one or more classes that it intends to extend. In other words, you are declaring inside your class, what other class’ methods do you want to be able to use, without having to strongly tie up your children class to its parent(s).

### Has-a VS Is-a

A good way to look at Composition is to think of a class that HAS-A relationship with another class, rather IS-A “son” of that class. In this sense, when you use Composition, the class has an “identity” of its own, and it’s not to be seen as being “only” a son of that other class which it extends.

## The Benefits of Composition

While I am not advocating for the complete elimination of traditional inheritance, there are benefits of using Composition that I would like to highlight. And, the way I see it, it is possible that once we realise the power of Composition over Inheritance, we may actually end up using Inheritance less and less. Here are the reasons why Composition is a good pattern:

1. **Flexibility**: it is not by chance that I am giving this one the first position in my list. With Inheritance, as mentioned, you tie your Class up to another one, and you define even before starting to design your class, who is going to extend (be a child of) what. But, what happens if you wanted to only use some of the methods that your parent has? Or, on the opposite side, what if you had 5 children of which only 2 needed a couple of additional methods? These things would not have been a problem with Composition. But the truth is that **Inheritance is often not flexible enough**.
2. **Extension of Multiple Classes**: who says you should only extend one Class? Why couldn’t Corolla be a son of Car but also of Toyota at the same time? That obviously makes sense, but it’s normally not possible in traditional programming languages. [Interfaces](http://en.wikipedia.org/wiki/Interface_(computing)#Software_interfaces_in_object-oriented_languages) can give us some help here, but they don’t actually implement the methods we need.
3. **Avoid duplicates**: Like in the previous example, if we were to use Interfaces for ourCorolla Class, we would end up having to write some of our methods for our Yaris or SedanClasses. That’s not cool. Duplicating code is not a good idea.

There are other benefits to using Composition, but those are the ones I feel are the most worthy of being highlighted here.