The Secret Alchemy of Object Composition

*“Object Composition Assembling or composing objects to get more complex behavior.” ~ Gang of Four,*

[*“Design Patterns: Elements of Reusable Object-Oriented Software”*](https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=as_li_ss_tl?ie=UTF8&linkCode=ll1&tag=eejs-20&linkId=06ccc4a53e0a9e5ebd65ffeed9755744)

*“Favor object composition over class inheritance.” ~ Gang of Four, “Design Patterns”.*

*“Favor a good design over the forced use of stylist elements like Composition and Inheritance” ~ The Prof*

One of the most common mistakes in software development is the tendency to overuse class inheritance. Class inheritance is a code reuse mechanism where instances form **is-a** relations with base classes. If you’re tempted to model your domain using *is-a* relations (e.g., a duck *is-a* bird) you’re bound for trouble, because class inheritance is the tightest form of coupling available in object-oriented design, which leads to many common problems, including (among others):

* The fragile base class problem
* The gorilla/banana problem
* The duplication by necessity problem

Class inheritance accomplishes reuse by abstracting a common interface away into a base class that subclasses can inherit from, add to, and override. There are two important parts of **abstraction**:

* **Generalization** The process of extracting only the shared properties and behaviors that serve the general use case
* **Specialization** The process of providing the implementation details required to serve the special case

There are lots of ways to accomplish generalization and specialization in code. Some good alternatives to class inheritance include simple functions, higher order functions, and *object composition*.

**Unfortunately, object composition is very misunderstood, and many people struggle to think in terms of object composition. It’s time to explore the topic in a bit more depth.**

**What is Object Composition?**

***“In computer science, a composite data type or compound data type is any data type which can be constructed in a program using the programming language’s primitive data types and other composite types. […] The act of constructing a composite type is known as composition.” ~ Wikipedia***

**One of the reasons for the confusion surrounding object composition is that any assembly of primitive types to form a composite object is a form of object composition, but inheritance techniques are often discussed in contrast to object composition as if they are different things. The reason for the dual meaning is that there is a difference between the grammar and semantics of object composition.**

When discussing object composition vs class inheritance, we’re not talking about specific techniques: We’re talking about the *semantic relationships* and *degree of coupling* between the component objects. We’re talking about *meaning* as opposed to *grammar*. People often fail to make the distinction and get mired in the grammar details. They can’t see the forest for the trees.

There are many different ways to compose objects. Different forms of composition will produce different composite structures and different relationships between the objects. When objects depend on the objects they’re related to, those objects are coupled, meaning that changing one object could break the other.

The Gang of Four advice to “favor object composition over class inheritance” invites us to think of our objects as a composition of smaller, loosely coupled objects rather than wholesale inheritance from a monolithic base class. The GoF describes tightly coupled objects as “monolithic systems, where you can’t change or remove a class without understanding and changing many other classes. The system becomes a dense mass that’s hard to learn, port, and maintain.”

**Three Different Forms of Object Composition**

In “Design Patterns”, the Gang of Four states, “you’ll see object composition applied again and again in design patterns”, and goes on to describe various types of compositional relationships, including *aggregation* and *delegation*.

The authors of “Design Patterns” were primarily working with Smalltalk and C++ (later, examples were translated to Java). Building and changing object relations at runtime in those languages is a lot more complicated than it is in JavaScript, so they understandably did not include many details on the subject. However, no discussion of object composition in JavaScript would be complete without a discussion of dynamic object extension, aka *concatenation.*

For reasons of applicability to JavaScript and to form cleaner generalizations, we’ll diverge *slightly* from the definitions used in “Design Patterns”. For instance, we won’t require that aggregations *imply* control over subobject lifecycles. That simply isn’t true in a language with dynamic object extension.

Selecting the wrong axioms can unnecessarily restrict a useful generalization, and force us to come up with another name for a special case of the same general idea. Software developers don’t like to repeat ourselves when we don’t need to.

* **Aggregation** When an object is formed from an enumerable collection of subobjects. In other words, an object which *contains*other objects. Each subobject retains its own reference identity, such that it could be destructured from the aggregation without information loss, e.g., arrays, trees, etc.
* **Concatenation** When an object is formed by adding new properties to an existing object. Properties can be concatenated one at a time or copied from existing objects, e.g., jQuery plugins are created by concatenating new methods to the jQuery delegate prototype, jQuery.fn.
* **Delegation** When an object forwards or *delegates to* another object. e.g., [Ivan Sutherland’s Sketchpad](https://youtu.be/3wrn9cxlgls?t=250) (1962) (published in his [influential 1963 thesis](https://dspace.mit.edu/handle/1721.1/14979)) included instances with references to “masters” which were delegated to for shared properties. Photoshop includes “smart objects” that serve as local proxies which delegate to an external resource. JavaScript’s prototypes are also delegates: Array instances forward built-in array method calls to Array.prototype, objects to Object.prototype, etc...

It’s important to note that these different forms of composition are **not mutually exclusive.** It’s possible to implement delegation using aggregation, and class inheritance is implemented using delegation in JavaScript. Many software systems use more than one type of composition, e.g., jQuery’s plugins use concatenation to extend the jQuery delegate prototype, jQuery.fn. When client code calls a plugin method, the request is delegated to the method that was concatenated to the delegate prototype.

Note on code examples The code examples below will share the following setup code:

const objs = [  
 { a: 'a', b: 'ab' },  
 { b: 'b' },  
 { c: 'c', b: 'cb' }  
];

**Aggregation**

Aggregation is when an object is formed from an enumerable collection of subobjects. An aggregate is an object which *contains*other objects. Each subobject in an aggregation retains its own reference identity, and could be losslessly destructured from the aggregate. Aggregates can be represented in a wide variety of structures.

**Examples**

* Arrays
* Maps
* Sets
* Graphs
* Trees
* DOM nodes (a DOM node may *contain* child nodes)
* UI components (a component may *contain* child components)

**When to use**

Whenever there are collections of objects which need to share common operations, such as iterables, stacks, queues, trees, graphs, state machines, or the composite pattern (when you want a single item to share the same interface as many items).

**Considerations**

Aggregations are great for applying universal abstractions, such as applying a function to each member of an aggregate (e.g., array.map(fn)), transforming vectors as if they're single values, and so on. If there are potentially hundreds of thousands or millions of subobjects, however, stream processing may be more efficient.

**Code examples**

Array aggregation:

const collection = (a, e) => a.concat([e]);const a = objs.reduce(collection, []);console.log(   
 'collection aggregation',  
 a,  
 a[1].b,  
 a[2].c,  
 `enumerable keys: ${ Object.keys(a) }`  
);

This will produce:

collection aggregation  
[{"a":"a","b":"ab"},{"b":"b"},{"c":"c","b":"cb"}]  
b c  
enumerable keys: 0,1,2

Linked list aggregation using pairs:

const pair = (a, b) => [b, a];const l = objs.reduceRight(pair, []);console.log(  
 'linked list aggregation',  
 l,  
 `enumerable keys: ${ Object.keys(l) }`  
);/\*  
linked list aggregation  
[  
 {"a":"a","b":"ab"}, [  
 {"b":"b"}, [  
 {"c":"c","b":"cb"},  
 []  
 ]  
 ]  
]  
enumerable keys: 0,1  
\*/

Linked lists form the basis of lots of other data structures and aggregations, such as arrays, strings, and various kinds of trees. There are many other possible kinds of aggregation. We won’t cover them all in-depth here.

**Concatenation**

Concatenation is when an object is formed by adding new properties to an existing object.

**Examples**

* Plugins are added to jQuery.fn via concatenation
* State reducers (e.g., Redux)
* Functional mixins

When to use: Any time it would be useful to progressively assemble data structures at runtime, e.g., merging JSON objects, hydrating application state from multiple sources, creating updates to immutable state (by merging previous state with new data), etc…

**Considerations**

* Be careful mutating existing objects. Shared mutable state is a recipe for many bugs.
* It’s possible to mimic class hierarchies and is-a relations with concatenation. The same problems apply. Think in terms of composing small, independent objects rather than inheriting props from a “base” instance and applying differential inheritance.
* Beware of implicit inter-component dependencies.
* Property name collisions are resolved by concatenation order: last-in wins. This is useful for defaults/overrides behavior, but can be problematic if the order shouldn’t matter.

const concatenate = (a, o) => ({...a, ...o});const c = objs.reduce(concatenate, {});console.log(  
 'concatenation',  
 c,  
 `enumerable keys: ${ Object.keys(c) }`  
);// concatenation { a: 'a', b: 'cb', c: 'c' } enumerable keys: a,b,c

**Delegation**

Delegation is when an object forwards or *delegates to* another object.

**Examples**

* JavaScript’s built-in types use delegation to forward built-in method calls up the prototype chain. e.g., [].map() delegates to Array.prototype.map(), obj.hasOwnProperty() delegates to Object.prototype.hasOwnProperty() and so on.
* jQuery plugins rely on delegation to share built-in and plugin methods among all jQuery object instances.
* Sketchpad’s “masters” were dynamic delegates. Modifications to the delegate would be reflected instantly in all of the object instances.
* Photoshop uses delegates called “smart objects” to refer to images and resources defined in separate files. Changes to the object that smart objects refer to are reflected in all instances of the smart object.

**When to use**

1. Conserve memory: Any time there may be potentially many instances of an object and it would be useful to share identical properties or methods among each instance which would otherwise require allocating more memory.
2. Dynamically update many instances: Any time many instances of an object need to share identical state which may need to be updated dynamically and changes instantaneously reflected in every instance, e.g., Sketchpad’s “masters” or Photoshop’s “smart objects”.

**Considerations**

* Delegation is commonly used to imitate class inheritance in JavaScript (wired up by the extends keyword), but is very rarely actually needed.
* Delegation can be used to exactly mimic the behavior and limitations of class inheritance. In fact, class inheritance in JavaScript is built on top of static delegates via the prototype delegation chain. Avoid *is-a* thinking.
* Delegate props are non-enumerable using common mechanisms such as Object.keys(instanceObj).
* Delegation saves memory at the cost of property lookup performance, and some JS engine optimizations get turned off for dynamic delegates (delegates that change after they’ve been created). However, even in the slowest case, property lookup performance is measured in millions of ops per second — chances are good that this is not your bottleneck unless you’re building a utility library for object operations or graphics programming, e.g., RxJS or three.js.
* Need to differentiate between instance state, and delegate state.
* Shared state on dynamic delegates is not instance safe. Changes are shared between all instances. Shared state on dynamic delegates is commonly (but not always) a bug.
* ES6 classes don’t create reassignable delegates in ES6. They may seem to work in Babel, but will fail hard in real ES6 environments.

**Code example**

const delegate = (a, b) => Object.assign(Object.create(a), b);  
  
const d = objs.reduceRight(delegate, {});  
  
console.log(  
 'delegation',  
 d,  
 `enumerable keys: ${ Object.keys(d) }`  
);  
  
// delegation { a: 'a', b: 'ab' } enumerable keys: a,b  
  
console.log(d.b, d.c); // ab c

**Conclusion**

We have learned:

* All objects made from other objects and language primitives are *composite objects*.
* The act of creating a composite object is known as *composition*.
* There are different kinds of object composition.
* The relationships and dependencies we form when we compose objects differ depending on how objects are composed.
* *Is-a relations* (the kind formed by class inheritance) are the tightest form of coupling in OO design, and should generally be avoided when its practical.
* The Gang of Four admonishes us to compose objects by assembling smaller features to form a larger whole, rather than inheriting from a monolithic base class or base object. “Favor object composition over class inheritance.”
* Aggregation composes objects into enumerable collections where each member of the collection retains its own identity, e.g., arrays, DOM tree, etc…
* Delegation composes objects by linking together an object delegation chain where an object forwards or delegates property lookups to another object. e.g., [].map() delegates to Array.prototype.map()
* Concatenation composes objects by extending an existing object with new properties, e.g., Object.assign(destination, a, b), {...a, ...b}.
* The definitions of different kinds of object composition are not mutually exclusive. Delegation is a subset of aggregation, concatenation can be used to form delegates and aggregates, and so on…

These are not the only three kinds of object composition. It’s also possible to form loose, dynamic relationships between objects through acquaintance/association relationships where objects are passed as parameters to other objects (dependency injection), and so on.

All software development is composition. There are easy, flexible ways to compose objects, and brittle, arthritic ways. Some forms of object composition form loosely coupled relations between objects, and others form very tight coupling.

Look for ways to compose where a small change to program requirements would require only a small change to the code implementation.

Express your intention clearly and concisely, and remember: If you think you need class inheritance, chances are very good that there’s a better way to do it.