04 May 2017

Note to self: It is worth digging into this one deeply, to understand the pros and cons of inheritance vs composition in different contexts.

|  |  |  |
| --- | --- | --- |
|  | Inheritance | Composition |
| Benefit |  | Change behavior on the fly. |
| Cost | More rigid because most languages allow only single class inheritance. |  |
| When to Use | TypeB wants to expose the \*complete\* interface of TypeA. | TypeB wants to expose \*parts of the\* interface of TypeA. |

## Liskov from 1999 MIT Article by Liskov and Wang

Introduction. Reviewed :-)

* Users of a type must not be able to tell that we swapped in a subtype of it.
* … if the program performs correctly with type T,
* … then the program must perform correct with type S
* … where type S is a subtype if T
* Contra/covariance rules are not enough to guarantee this.
* … those are syntactic rules not semantic rules
* … the compiler will catch a breakage
* … but the semantic might break the program
* … e.g. Stack vs Queue could both have a `put` method
* The provable properties of a type must be present on its subtype.
* The concern is only with "safety properties"
* … "nothing bad happens"
* … 1. Invariants, which are properties true of all states (single states)
  + A bag's size is always less than its bound.
* … 2. History properties, which are properties true of all sequences of states (pairs of states)
  + A bag's bound does not change.
* Liskov's Substitution Principle
  + Handles mutable types
  + Allows subtypes to have more methods than their supertype.

Motivation

Model of Computation

Specifications

.. Type

.. Creators

.. Explicit Invariants

.. Explicit Constraints

The Meaning of Subtype

.. Specifying

.. Defining

.. Checklist

Type Hierarchies

.. Extension Subtypes

.. Constrained Subtypes

Related Work

Summary

## Liskov from Wikipedia

Liskov Substitution Principle <https://en.wikipedia.org/wiki/Liskov_substitution_principle>

Aka suitability

* If S is a subtype of T
* …then we can replace object of type T
* ...with object of type S
* …without altering the desirable properties of T.

Aka strong behavioral subtyping

* Liskov (1987) keynote and Liskov, Wing (1994) paper. The latter updated in 1999.
* It is a semantic not a syntactic relationship between types,
* … if p(x) is a property
* … that is provable
* … for objects x of type T
* … then p(y) should be true
* … for objects y of type S
* … where is a subtype of T

## Variance

Variance: refers to how the subtyping of complex types relates to subtyping of that type's components. The ordering of the types can be preserved, reversed, ignored, or include both directions. These are covariances, contravariance, invariance, and bivariance respectively.

|  |
| --- |
| Variance  See also: https://en.wikipedia.org/wiki/Covariance\_and\_contravariance\_(computer\_science) |
| using System.Collections.Generic;  public class Program  {  public static void Main (string[] args) {    // component ordering of types: string is an object  object obj = string.Empty;    // covariance: preserves the ordering of types  IEnumerable<object> objects = new List<string>();    // contravariance: reverses the ordering of types  Action<string> doString = (Action<object>)(s => s.ToString());  }  } |

## Wat

Important: Inheritance is not the same as implementation!

* [What do I mean by this?]
* Inheritance happens with classes.
* Implementation happens with abstractions (interfaces.)

## Composition over Inheritance

<http://wiki.c2.com/?CompositionInsteadOfInheritance> This is a dense article that has a lot of nuances! It honors the complexity of the composition vs inheritance decision by resisting simple rules.

* Is-a vs has-a decision.
* A person is a mouth vs a person has a mouth.
  + Beware breaking the Law of Demeter.
  + <http://wiki.c2.com/?LawOfDemeter>
  + I.e. Use person.eat() not person.getMouth().eat().
* Composition exposes violations of the Law of Demeter.
* What about a Window?
  + Has a Rectangle?
  + Is a Rectangle?
  + Has a Shape!
  + Rectangle is a Shape.
* Something to contemplate:
* … Inheritance is for exposing the interface of a lower-level class from a higher-level class.
* … Composition is for using the interface of a lower-level class in a higher-level class.
* The main downside of composition are
  + "more code volume" aka more typing,
  + Greater risk of breaking the Law of Demeter.
* The main upsides are
  + Easier refactoring
  + Able to change at runtime.
* It's important to note that both techniques are useful; though, may people find composition to be appropriate for more use cases than inheritance is.

<https://en.wikipedia.org/wiki/Composition_over_inheritance>

* In progress\*
* This has some references worth reading. E.g. Tempero (2013)
* Aka Composite Reuse Principle
* How?
  + First, define interfaces.
  + Second, build classes that implement the interfaces.
  + Third, define business domain classes composed of those interfaces.
  + Business objects often implement the interfaces of which they are also composed. E.g.
    - Person might implement IAnimal
    - ...and take an instance of an IAnimal in the constructor.
* Why?
  + By using a different implementation of an interface,
  + ...we can change the composition of business domain classes,
  + ...even as late as at runtime.
* Aside:
  + Mentions the **Diamond Problem**,
  + …which occurs with multiple inheritance,
  + …which occurs when using inheritance instead of composition.
* Benefits
  + Greater flexibility
  + More natural way to design
  + More stable business domain
* Aside:
  + New term: **transparency.**
* Downsides:
  + Forwarding methods, in which method Foo() calls component.Foo().
  + Some languages have features that handle this nicely.
  + Traits, mixins, protocol extensions.

<https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose>

* Both inheritance and composition are for code reuse.
* Composition is about making things out of parts.
* Inheritance is about two things:
  + Semantics – Reuse of the meaning of the parent. (is-a or sub-type).
  + Mechanics – Reuse of the implementation of the parent.

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

* Downsides of inheritance:
  + The subtype is tightly coupled to its parent.

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