Why is inheritance only defined at compile-time?

Asked 8 years, 11 months ago Modified 8 years, 11 months ago Viewed 5k times

7

I found this statement from the gang of four's "Design Patterns" particularly odd; for some context, the authors are comparing inheritance versus composition as reuse mechanisms [p. 19]:

> "...you can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time."

They go on to say of composition:

> "Object composition is defined dynamically at run-time through objects acquiring references to other objects."

I am not sure why this phase distinction is important. I am familiar with compiling and inheritance but work as a JavaScript developer, so maybe I'm missing something fundamental.

design-patterns inheritance compilation runtime

Share Improve this question Follow

asked Jan 21, 2014 at 18:29

jds
1,092 2 9 19

9 *Design Patterns* mainly assumes C++, where this distinction is true. In more dynamic languages like Lisp, Smalltalk, Perl, Python, Ruby, Javascript this distinction between runtime and compile-time does not exist or is blurred. Classes can be created and/or modified at "runtime". However, patterns assuming these dynamic features would not be useful to more static languages like C++ or Java which are arguably more important. – amon Jan 21, 2014 at 18:42

So that inheritance errors could be caught at compile-time. – Den Dec 1, 2015 at 10:50

3 Answers

Sorted by: Highest score (default)

7

Some languages are pretty strongly static, and only allow the specification of the inheritance relationship between two classes at the time of definition of those classes. For C++, definition time is practically the same as compilation time. (It's *slightly* different in

Java and C#, but not very much.) Other languages allow much more dynamic reconfiguration of the relationship of classes (and class-like objects in Javascript) to each other; some go as far as allowing the class of an existing object to be modified, or the superclass of a class to be changed. (This can cause *total* logical chaos, but can also model real world nasties quite well.)

But it is important to contrast this to composition, where the relationship between one object and another is not defined by their class relationship (i.e., their *type*) but rather by the references that each has in relation to the other. General composition is a very powerful and ubiquitous method of arranging objects: when one object needs to know something about another, it has a reference to that other object and invokes methods upon it as necessary. As soon as you start looking for this *super-fundamental* pattern, you'll find it absolutely everywhere; the only way to avoid it is to put everything in one object, which would be massively dumb! (There's also stricter UML composition/aggregation, but that's not what the GoF book is talking about there.)

One of the things about the composition relationship is that particular objects do not need to be hard-bound to each other. The pattern of concrete objects is very flexible, even in very static languages like C++. (There is an upside to having things very static: it is possible to analyse the code more closely and — at least potentially — issue better code with less overhead.) To recap, Javascript, as with many other dynamic languages, can pretend it doesn't use compilation at all; just pretence, of course, but the fundamental language model doesn't require transformation to a fixed intermediate format (e.g., a “binary executable on disk”). That compilation which is done is done at runtime, and can be easily redone if things vary too much. (The *fascinating* thing is that such a good job of compilation can be done, even starting from a very dynamic basis...)

Some GoF patterns only really make sense in the context of a language where things are fairly static. That's OK; it just means that not all forces affecting the pattern are necessarily listed. One of the key points about studying patterns is that it helps us be aware of these important differences and caveats. (Other patterns are more universal. Keep your eyes open for those.)

Share Improve this answer Follow

answered Jan 21, 2014 at 21:47

Donal Fellows
6,327 25 35

1

In Javascript, there seemingly isn't much difference, because there is no compilation. From what I read, what's commonly called inheritance in JS is just creating a prototype chain between different objects.
But in languages like C#, you can create an inheritance hierarchy at compile time through classes inheriting from each other, but you can also change behavior of objects at runtime through patterns like the strategy pattern.

Share Improve this answer Follow

answered Jan 21, 2014 at 18:42

Stefan Billiet
3,499 1 17 14

1 Am I reading it wrong, or is this a circular argument? "Inheritance is only defined at compile time" because "in languages like C#, you can create an inheritance hierarchy at compile time"? – user39685 Jan 22, 2014 at 14:07

It's not really correct to say that "there's no compilation in Javascript". Plus, it's pretty vague -- perhaps it's intended to mean that there's no static type-checking? – user39685 Jan 22, 2014 at 14:10

1

Let's consider a strongly typed language like C#. Here you would define inheritance like this:

```
class BaseClass
{
    public string SomeMethod()
    {
        return "Hello";
    }
}

class DerivedClass : BaseClass  // Inherits BaseClass
{
    public string AdditionalMethod()
    {
        return "World";
    }
}
```

Usage:

```
var d = new DerivedClass();
Console.WriteLine(d.SomeMethod()); // --> Hello
Console.WriteLine(d.AdditionalMethod()); // --> World
```

The class `DerivedClass` inherits the method `SomeMethod` from `BaseClass`. This fact is defined at compile time. Sometimes the term "design time" is used instead. The latter term makes clear that the inheritance is already defined in the code text by the programmer.

In contrast, let's look how composition works

```
interface IReturnsString
{
    string GetString();
}

class ComposedClass
{
    private IReturnsString _a, _b;

    public ComposedClass(IReturnsString a, IReturnsString b) // Constructor
    {
        _a = a;
        _b = b;
    }
```

```
    public string SomeMethodA()
    {
        return _a.GetString();
    }

    public string SomeMethodB()
    {
        return _b.GetString();
    }

    public void ChangeBehavior()
    {
        _a = new ClassC(); // ClassC is supposed to implement IReturnsString.
    }
}
```

At runtime you can pass different implementations to the composed class. The composed class could even choose to change its behavoir at runtime.

```
var x = new ComposedClass(new ClassA(), new ClassB());
Console.WriteLine(x.SomeMethodA()); // Prints something

x.ChangeBehavior();
Console.WriteLine(x.SomeMethodA()); // Prints something else

var y = new ComposedClass(new ClassD(), new ClassE());
// y behaves in a different way than x
```

Share Improve this answer

Follow

edited Jan 22, 2014 at 13:24

answered Jan 21, 2014 at 21:37

Olivier Jacot-Descombes

1,664 10 14

x.SomeMethodB() shouldn't print something else because only SomeMethodA() is affected by ChangeBehavior(). Very good example. I don't think I would have spotted it without the comments. – JeffO Jan 21, 2014 at 23:11

s/strong/static/ – user39685 Jan 22, 2014 at 14:04