What are some advantages of duck-typing vs. static typing?

Asked 16 years, 3 months ago Modified 6 years, 8 months ago

Viewed 15k times

33

I'm researching and experimenting more with Groovy and I'm trying to wrap my mind around the pros and cons of implementing things in Groovy that I can't/don't do in Java. Dynamic programming is still just a concept to me since I've been deeply steeped static and strongly typed languages.

Groovy gives me the ability to duck-type, but I can't really see the value. How is duck-typing more productive than static typing? What kind of things can I do in my code practice to help me grasp the benefits of it?

I ask this question with Groovy in mind but I understand it isn't necessarily a Groovy question so I welcome answers from every code camp.

groovy duck-typing

Share

Improve this question

Follow

edited Feb 10, 2009 at 16:41

asked Sep 7, 2008 at 0:28

codeLes

3,069 ● 3 ● 29 ● 27

2 Duck typing and static typing are orthogonal concepts. – J D Dec 14, 2010 at 15:53

10 Answers

Sorted by: Highest score (default)

25

A lot of the comments for duck typing don't really substantiate the claims. Not "having to worry" about a type is not sustainable for maintenance or making an application extendable. I've really had a good opportunity to see Grails in action over my last contract and its quite funny to watch really. Everyone is happy about the gains in being able to "create-app" and get going - sadly it all catches up to you on the back end.

Groovy seems the same way to me. Sure you can write very succinct code and definitely there is some nice sugar in how we get to work with properties, collections, etc... But the cost of not knowing what the heck is being passed back and forth just gets worse and worse. At some point your scratching your head wondering why the project has become 80% testing and 20% work. The lesson here is that "smaller" does not make for "more readable" code. Sorry folks, its simple logic - the more you have to know intuitively then the more complex the process of understanding that code becomes. It's why GUI's have backed off becoming overly iconic over the

years - sure looks pretty but WTH is going on is not always obvious.

People on that project seemed to have troubles "nailing down" the lessons learned, but when you have methods returning either a single element of type T, an array of T, an ErrorResult or a null ... it becomes rather apparent.

One thing working with Groovy has done for me however - awesome billable hours woot!

Share Improve this answer Follow

edited Jan 18, 2010 at 23:15

answered Jan 18, 2010 at 23:10

Lypheus

474 ● 7 ● 11

1 this is the most practical answer ever – abchau Nov 16, 2022 at 20:11

13

Duck typing cripples most modern IDE's static checking, which can point out errors as you type. Some consider this an advantage. I want the IDE/Compiler to tell me I've made a stupid programmer trick as soon as possible.

My most recent favorite argument **against** duck typing comes from a Grails project DTO:

```
class SimpleResults {
    def results
```

```
    def total
    def categories
}
```

where `results` turns out to be something like `Map<String, List<ComplexType>>`, which can be discovered only by following a trail of method calls in different classes until you find where it was created. For the terminally curious, `total` is the sum of the sizes of the `List<ComplexType>`s and `categories` is the size of the `Map`

It may have been clear to the original developer, but the poor maintenance guy (ME) lost a lot of hair tracking this one down.

Share Improve this answer

Follow

answered Oct 30, 2008 at 13:09

Ken Gentle

13.3k ● 2 ● 43 ● 49

1 Although explicit type declarations can be instructive, reflection is often a decent substitute. Open an interactive session and ask those objects what type they are. – Nick Retallack Feb 16, 2010 at 23:53

1 Your example points out the problems with poor documentation, not with dynamic typing (and there's no duck typing in sight here). Under static typing I would know (because it's declared) that `results` is a `Map<String, List<ComplexType>>`, and that `total` is an `int` and `categories` is an `int`. That doesn't tell me me much about how to use this class. – Ben Jan 29, 2012 at 1:25

1 The problem with this code is not typing but the fact that those are non-descriptive reference names. If the developer

had called results something expressive like mapOfFooToListOfBar then you would have never had this problem. Naming just about anything results or total is a code smell and the mark of a poor craftsman. This code also suffers from primitive obsession. – jeremyjjbrown Apr 15, 2013 at 3:10

2 @jeremyjjbrown, of course it's a code smell, unprofessional and the mark of a poor craftsman - but it "works" solely because of "duck typing" in Groovy. Yes, descriptive names would help. Remember, we're talking about the poor schmuck who has to pick this up after the developer is long gone, and even having the long names isn't enough to know how to manipulate the content of *results* WITHOUT FURTHER INVESTIGATION (read that: time, money and resources) – Ken Gentle Apr 15, 2013 at 14:53

1 @ Ken, "how to manipulate the content of results". So I assume there are no tests either. I'm having the same exact problem your having right now (today) with stinky code with no tests, written in plain Java. Static typing hasn't added to it's maintainability at all. But if what you are saying is that Duck typing allows developers to be even more irresponsible, then I totally agree with you. – jeremyjjbrown Apr 15, 2013 at 17:38

10

It's a little bit difficult to see the value of duck typing until you've used it for a little while. Once you get used to it, you'll realize how much of a load off your mind it is to not have to deal with interfaces or having to worry about exactly what type something is.

Share Improve this answer Follow

answered Sep 7, 2008 at 0:42


Jason Baker
198k ● 138 ● 382 ● 520




Next, which is better: EMACS or vi? This is one of the running religious wars.

Think of it this way: any program that is *correct*, will be correct if the language is statically typed. What static typing does is let the compiler have enough information to detect type mismatches at compile time instead of run time. This can be an annoyance if your doing incremental sorts of programming, although (I maintain) if you're thinking clearly about your program it doesn't much matter; on the other hand, if you're building a really big program, like an operating system or a telephone switch, with dozens or hundreds or thousands of people working on it, or with really high reliability requirements, then having he compiler be able to detect a large class of problems for you without needing a test case to exercise just the right code path.

It's not as if dynamic typing is a new and different thing: C, for example, is effectively dynamically typed, since I can always cast a `foo*` to a `bar*`. It just means it's then my responsibility as a C programmer never to use code that is appropriate on a `bar*` when the address is really pointing to a `foo*`. But as a result of the issues with large programs, C grew tools like lint(1), strengthened its type system with `typedef` and eventually developed a strongly typed variant in C++. (And, of course, C++ in turn

developed ways around the strong typing, with all the varieties of casts and generics/templates and with RTTI.

One other thing, though --- don't confuse "agile programming" with "dynamic languages". Agile programming is about the way people work together in a project: can the project adapt to changing requirements to meet the customers' needs while maintaining a humane environment for the programmers? It can be done with dynamically typed languages, and often is, because they can be more productive (eg, Ruby, Smalltalk), but it can be done, has been done successfully, in C and even assembler. In fact, Rally Development even uses agile methods (SCRUM in particular) to do marketing and documentation.

Share Improve this answer

Follow

answered Nov 9, 2008 at 15:12

Charlie Martin

112k ● 26 ● 196 ● 264

15 It's not true that any program that is correct will be correct under static typing. Static typing is a conservative approximation to the runtime behavior. This is why some programs with casts can still be type correct (by preserving an invariant which the type checker is unable to prove). – Doug McClean Feb 10, 2009 at 16:48

8 Sorry, C's weak type system is *not* the same thing as dynamic typing. It's not even close. There is no late binding. Casting pointers like your example will cause the program to assume a different underlying structure leading to bugs, not functionality. – postfuturist Feb 10, 2009 at 16:49

Doug, I believe that's a theorem. Assume contrary: then you have a program you have a program which is correct, ie, meets postcondition, but where exists a statement for which no static typing is correct. But that's equivalent to exists a statement w/o defined semantics in correct prg, Contradicts – Charlie Martin Feb 10, 2009 at 21:11

Sorry, Steve, that's not correct either. Consider the case of a void* pointing to different structures. – Charlie Martin Feb 10, 2009 at 21:13

1 Two aspects of agile programming are plentiful unit tests and merciless refactoring. There's anecdotal evidence at least, that dynamically typed languages are helpful in both these areas. Having said that, static typing really helps with IDE-assisted refactoring. – slim Feb 17, 2010 at 4:43

7

There is nothing wrong with static typing if you are using Haskell, which has an incredible static type system. However, if you are using languages like Java and C++ that have terribly crippling type systems, duck typing is definitely an improvement.

Imagine trying to use something so simple as "map" in Java (and no, I don't mean the data structure). Even generics are rather poorly supported.

Share Improve this answer Follow

edited Feb 16, 2010 at 23:44

answered Sep 7, 2008 at 1:09

Nick Retallack
19.5k ● 19 ● 94 ● 115

1 Haskell does have a great type system -- it would be awesome if more languages utilized a similar mechanism. – mipadi Nov 9, 2008 at 16:15

There's nothing wrong with Java's Map type. The problem is Java's lack of a fluent syntax to deal with Maps. That's something Groovy addresses, but it's not really related to Groovy's duck typing. – slim Feb 16, 2010 at 12:50

@slim: I'm not talking about the 'map' type, I'm talking about the 'map' higher order function. I'll add a link to my answer. – Nick Retallack Feb 16, 2010 at 23:49

2 Oh, OK. In Groovy that's collect(): List doubled = x.collect { it * 2 }. Scala calls it map() and has strong typing. I don't see how Java's type system gets in the way of this. Java will get closures soon, and I'm sure the Collections framework will get collect/map. – slim Feb 17, 2010 at 4:34

Does anyone actually get to use Haskell at work? I'm not even sure some of the Java developers I work with could learn Haskell. – jeremyjjbrown Apr 15, 2013 at 3:19

▲

6

With, **TDD + 100% Code Coverage** + IDE tools to constantly run my tests, I do not feel a need of static typing any more. With no strong types, my unit testing has become so easy (Simply use Maps for creating mock

objects). Specially , when you are using Generics, you can see the difference:

```
//Static typing
Map<String,List<Class1<Class2>>> someMap = [:] as
HashMap<String,List<Class1<Class2>>>
```

vs

```
//Dynamic typing
def someMap = [:]
```

Share Improve this answer

Follow

answered Nov 17, 2013 at 8:30

sukrit007

671 ● 8 ● 10

5

IMHO, the advantage of duck typing becomes magnified when you adhere to some conventions, such as naming you variables and methods in a consistent way. Taking the example from **Ken G**, I think it would read best:

```
class SimpleResults {
    def mapOfListResults
    def total
    def categories
}
```

Let's say you define a contract on some operation named 'calculateRating(A,B)' where A and B adhere to another contract. In pseudocode, it would read:

```
Long calculateRating(A someObj, B, otherObj) {

   //some fake algorithm here:
   if(someObj.doStuff('foo') > otherObj.doStuff('bar')
someObj.calcRating());
   else return otherObj.calcRating();

}
```

If you want to implement this in Java, both A and B must implement some kind of interface that reads something like this:

```
public interface MyService {
    public int doStuff(String input);
}
```

Besides, if you want to generalize you contract for calculating ratings (let's say you have another algorithm for rating calculations), you also have to create an interface:

```
public long calculateRating(MyService A, MyServiceB);
```

With duck typing, you can ditch your **interfaces** and just rely that on runtime, both A and B will respond correctly to your `doStuff()` calls. There is no need for a specific contract definition. This can work for you but it can also work against you.

The downside is that you have to be extra careful in order to guarantee that your code does not break when some

other persons changes it (ie, the other person must be aware of the implicit contract on the method name and arguments).

Note that this aggravates specially in Java, where the syntax is not as terse as it could be (compared to Scala for example). A counter-example of this is the Lift framework, where they say that the SLOC count of the framework is similar to Rails, but the test code has less lines because they don't need to implement type checks within the tests.

Share Improve this answer

Follow

answered Nov 9, 2008 at 14:25

Miguel Ping

18.3k ● 23 ● 91 ● 137

3

Here's one scenario where duck typing saves work.

Here's a very trivial class

```
class BookFinder {
    def searchEngine

    def findBookByTitle(String title) {
         return searchEngine.find( [ "Title" : title ]
    }
}
```

Now for the unit test:

```
void bookFinderTest() {
    // with Expando we can 'fake' any object at runtim
```

```
    // alternatively you could write a MockSearchEngin
    def mockSearchEngine = new Expando()
    mockSearchEngine.find = {
        return new Book("Heart of Darkness","Joseph Co
    }

    def bf = new BookFinder()
    bf.searchEngine = mockSearchEngine
    def book = bf.findBookByTitle("Heart of Darkness")
    assert(book.author == "Joseph Conrad"
}
```

We were able to substitute an Expando for the SearchEngine, because of the absence of static type checking. With static type checking we would have had to ensure that SearchEngine was an interface, or at least an abstract class, and create a full mock implementation of it. That's labour intensive, or you can use a sophisticated single-purpose mocking framework. But duck typing is general-purpose, and has helped us.

Because of duck typing, our unit test can provide any old object in place of the dependency, just as long as it implements the methods that get called.

To emphasise - you can do this in a statically typed language, with careful use of interfaces and class hierarchies. But with duck typing you can do it with less thinking and fewer keystrokes.

That's an advantage of duck typing. It doesn't mean that dynamic typing is the right paradigm to use in all situations. In my Groovy projects, I like to switch back to

Java in circumstances where I feel that compiler warnings about types are going to help me.

Share Improve this answer

Follow

answered Feb 16, 2010 at 11:20

slim

41.2k ● 14 ● 99 ● 128

2 -1: You are talking about a specific kind of *nominal* static type system and assuming (incorrectly) that your observations apply to all static type systems. They do not – J D Dec 14, 2010 at 16:01

2

To me, they aren't horribly different if you see dynamically typed languages as simply a form of static typing where everything inherits from a sufficiently abstract base class.

Problems arise when, as many have pointed out, you start getting strange with this. Someone pointed out a function that returns a single object, a collection, or a null. Have the function return a specific type, not multiple. Use multiple functions for single vs collection.

What it boils down to is that anyone can write bad code. Static typing is a great safety device, but sometimes the helmet gets in the way when you want to feel the wind in your hair.

Share Improve this answer

Follow

answered Apr 18, 2018 at 6:53

Evan Langlois

4,345 ● 2 ● 21 ● 18

1

It's not that duck typing is more productive than static typing as much as it is simply different. With static typing you always have to worry that your data is the correct type and in Java it shows up through casting to the right type. With duck typing the type doesn't matter as long as it has the right method, so it really just eliminates a lot of the hassle of casting and conversions between types.

Share Improve this answer

Follow

answered Sep 7, 2008 at 0:31

Chris Bunch

89.6k ● 37 ● 129 ● 127

1 There is plenty of anecdotal evidence suggesting that duck typing is significantly more productive than static typing. – postfuturist Feb 10, 2009 at 16:55

1 @postfuturist: Only for nominal static type systems. – J D Dec 14, 2010 at 16:25

You say that duck typing is not more productive but then end by saying "it just eliminates a lot of the hassle of casting and conversions between types." I think it's quite clear that it's more productive (assuming you don't have to spend a ton of time debugging behind a programmer with poor practices), and your answer describes why quite well. – dallin Mar 9, 2022 at 5:35