Using the Single Responsibility Principle in the "real world" [closed]

Asked 15 years, 11 months ago Modified 10 years, 3 months ago

Viewed 4k times

19

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

Closed 13 years ago.

I basically want to get an idea of the percentage of people who think it's reasonable to use the Single Responsibility Principle in real-world code **and** how many actually do. In Podcast #38 Joel talks about how useless this OOP principle is the real world; and further that this demonstrates how people like Uncle Bob have likely not written non-trivial systems.

I've personally written or played a big role in a few software projects but have only now come across this pattern in my young career. I love the sound of this principle and would really like to start using it. I found Joel's argument in the podcast quite weak (as did others if you continue on to read the blog comments here). But, is there any truth in it?

What does the community think?

oop solid-principles single-responsibility-principle

Share

Improve this question

Follow

edited Jan 18, 2021 at 12:38

Community Bot

1 ● 1

asked Jan 28, 2009 at 18:05

Thiru

271 ● 2 ● 11

I think you should turn this into a wiki, as there is no single definite answer. – user1228 Jan 28, 2009 at 18:42

I'm not sure I agree. I think their are more and far less correct answers. And I think too many people believe this is far more subjective or context-sensitive than it actually is -- which is partly why I posted the question. Thanks for your comment though. – Thiru Jan 29, 2009 at 17:08

9 Answers

Sorted by: Highest score (default)

25

I have had some experience applying SOLID principles and my experience has been mainly good. I also heard the podcast and it sounds like neither Jeff nor Joel has tried any of the things they're talking about long enough to really asses the benefits. The main argument against is as usually "you write more code". If I look at what I do I write 10 maybe 20% more code (usually interface definitions) but because everything is highly decoupled it's far more maintainable. I hardly ever have situations where changes in one part of my application break other parts. So the 20% extra code i have to maintain pays for itself.

Jeff also missed the point on code quality. He doesnt see code quality as a great benefit for the customer. And he's right, the customer doesn't care. The customer does care about having new features implemented quickly and that's where code quality comes in. I've found that the investment of keeping the quality of code as high as possible has always paid itself back within a few months. High quality = low maintenance.

I do agree with them that like anything you have to be pragmatic about these things. If you need to deliver something then go ahead and do it fast and dirty. But clean up afterwards.

Share Improve this answer Follow

answered Jan 28, 2009 at 18:24

Mendelt

37.5k ● 6 ● 75 ● 97

8

I'm working on a project that does lots of different, horribly complex things, in a framework that is supposed to be easily extensible by others.

At first, classes were large and did multiple things. In order to change those behaviors, you had to extend those classes. The methods were all virtual and didn't change the object's state, so it was pretty easy to do this.

But as the system grew, it became more clear that the framework was going to end up with a series of monolithic objects, each with looooong inheritance chains. This also resulted in unexpected dependencies--an abstract method taking a collection of class X to produce object Y defined in a base class dictated that EVERYBODY had to do it this way, even when it made no sense for half of the inheritance tree. This also resulted in massive classes that required tens of unit tests to get the code coverage over 80%, and the complexity was such that you weren't sure if you covered everything correctly. It was obvious this design would cause the framework to be very rigid and inflexible.

So we redesigned everything along SRP lines. You'd have your interface, a base class and, possibly, one or more implementation classes. Each one was *composed* of different objects which performed key functions of the overall process. If you wanted to change one part, you didn't override a method, you would produce another object that extended the requisite interface/base class and performed its job differently. SRP was even taken down into the arguments and return values of the class methods. For those parts of the system that needed to be flexible, rather than pass collections of X class that are used to produce Y objects, a class was created to encapsulate the process of production of Y objects. Components in the system would then pass these producers around, combine them with others (the responsibility of the producers), and eventually use them to produce Y. This allowed for different types of producers to be created, all of which could be treated exactly the same, even though they did vastly different things. The move also drastically reduced each class' code base and made them MUCH easier to test.

I would say that, as a new developer, it is VERY HARD to break everything down to this level. You almost have to write a big ball of mud, understand how it works, then redesign it as several different components, each taking responsibility for a part of the whole.

Share Improve this answer Follow

answered Jan 28, 2009 at 18:37

user1228

5

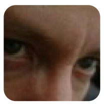
The biggest problem with some of the large number of programming theories out there is that they focus on suggesting that good attributes in code, are in fact good attributes for code. They're right because they are intrinsically right, and thus not particularly useful.

Yes, code should be well written, and yes, things shouldn't be horribly repeated, and yes, changes shouldn't cause weird breaks in unexpected places. But, at the end of the day, really simple, really consistent code that expresses the solution in a simple, easily understandable way is worth far more than a big complex system that meets some rigorous principles. As an industry we tend to take good ideas way too far, creating masses of unnecessary complexity in the quest for the "right" solution, not the best one.

Paul.

Share Improve this answer Follow

answered Jan 28, 2009 at 20:36



Paul W Homer

2,740 ● 1 ● 19 ● 25

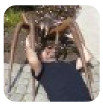
4

I haven't read or listened to Joel's comments so can't comment on those specifically.

I think you have to look at the single responsibility principle in terms of a goal, rather than a strict requirement. As with many things in software development, there are ideals that should be strived for but, unless your code has no monetary benefit or cost, you have to be pragmatic in serving your customer's needs.

Share Improve this answer Follow

answered Jan 28, 2009 at 18:12



BlackWasp

4,961 ● 2 ● 33 ● 42

I doubt if any customer cares about developer discipline and enforcement of OO principles, but you (or anybody else) that has to maintain/work with the code may appreciate it.
– Jim Anderson Jan 28, 2009 at 18:43

I don't disagree. It's jsut that sometimes pragmatism wins over strict enforcement of this particular principle.
– BlackWasp Jan 29, 2009 at 15:25

4

In practice, it's very hard to get true SRP in OO situations. Consider a class that exists in a complicated system. It probably has only one business-oriented responsibility (such as printing a report), but it will have many other responsibilities internally that violate pure SRP ideals, such as logging and exception handling. If you change your logging mechanism significantly, you'll probably need to change the calls your printing class makes.

This is why AOP was conceived. Using it you don't have to change anything but the logging classes to change logging.

It's good to strive for business-oriented SRP for obvious reasons, but for sufficiently complicated systems you'll never get true SRP in OOP. AOP, sure, but not OOP.

I have no idea if that's what Joel's reasoning is, but that's how I would approach the idea.

Share Improve this answer Follow

answered Jan 28, 2009 at 18:15
Welbog
60.3k ● 9 ● 113 ● 124

I think the biggest benefits to being a disciplined OO developer and adhering to the SRP whenever possible

1

are ease of testability and maintenance so I would say SRP is a good goal for any system that is going to be tested/maintained (basically, anything except a throwaway system).

answered Jan 28, 2009 at 18:46
Jim Anderson
3,622 ● 2 ● 26 ● 21

1

I think SOLID principles sometimes don't comply with natural/business logic that one usually applies when designing classes. Robert C. Martin brought up an example of Rectange and Square classes (Square shouldn't be a descendant of Rectangle).

http://www.hanselminutes.com/default.aspx?showID=163

I'm not sure that it was regarding SRP but the idea is the same. SOLID principles may lead to counterintuitive decisions and it's hard to get though this barrier.

To me 'guidelines' is more appropriate name than 'principles'.

answered Feb 8, 2009 at 3:57
Bogdan Kanivets
562 ● 2 ● 9 ● 17

In general I agree with the SOLID principles, but you also have to take them into context. If you are writing a Proof

0

Of Concept, then the SOLID principles are less applicable.

On the other hand if you are developing a product with a life that will span years, then you better look carefully at the SOLID principles and apply them. Otherwise it's going to cost the company tons of money in productivity.

In regards to Joel and his comments, he has a valid point. Sometimes the product needs to ship or the company fails. That's just the way it is.

As a developer it's your job to ship the product, but just because there is a tight deadline doesn't mean you go with a poor architecture. Choices like do you use datasets or business entities is a simple decision and both have similar implementation efforts, but in the long term the new feature development and maintenance between the two is drastic.

Share Improve this answer Follow

answered Jan 28, 2009 at 18:23

Chuck Conway
16.4k ● 11 ● 61 ● 102

0

I think that it should be possible to write a simple one-line main responsibility for all classes. The word "and" in this one-liner is usually not a good sign. The single responsibility then often boils down to selecting the proper abstraction.

GUI near classes tend to reflect GUI and have the single responsibility of "being gui for XXXX". Coward's solution..

Share Improve this answer

Follow

answered Jan 28, 2009 at 18:38

krosenvold

77k ● 33 ● 156 ● 209