It this an example of the Single Responsibility Principle?

Asked 15 years, 9 months ago Modified 14 years, 2 months ago Viewed 6k times

6

I made the following code example to learn how to use a generics method signature.

In order to get a **Display() method** for both Customer and Employee, I actually began replacing my **IPerson interface** with an **Person abstract class**.

But then I stopped, remembering a podcast in which Uncle Bob was telling Scott Hanselman about the **Single Responsibility Principle** in which you should have lots of little classes each doing one specific thing, i.e. that a Customer class should not have a **Print()** and **Save()** and **CalculateSalary()** method but that you should have a ***CustomerPrinter class*** and a ***CustomerSaver class*** and a ***CustomerSalaryCalculator class***.

That seems an odd way to program. However, **getting rid of my interface also felt wrong** (since so many IoC containers and DI examples use them inherently) so I decided to give the Single Responsibility Principle a try.

So **the following code is different than I have programmed in the past** (I would have made an abstract class with a Display() method and got rid of the interface) but based on what I have heard about decoupling and the S.O.L.I.D. principles, **this new way of coding** (the interface and the PersonDisplayer class) **I think this is the right way to go**.

I would **like to hear if others think the same way** on this issue or have experienced positive or negative effects of this (e.g. an unwieldy number of classes each doing one particular thing, etc.).

```
using System;

namespace TestGeneric33
{
    class Program
    {
        static void Main(string[] args)
        {
            Container container = new Container();
            Customer customer1 = container.InstantiateType<Customer>("Jim",
"Smith");
            Employee employee1 = container.InstantiateType<Employee>("Joe",
"Thompson");
            Console.WriteLine(PersonDisplayer.SimpleDisplay(customer1));
            Console.WriteLine(PersonDisplayer.SimpleDisplay(employee1));
            Console.ReadLine();
        }
    }

    public class Container
    {
        public T InstantiateType<T>(string firstName, string lastName) where T
: IPerson, new()
        {
            T obj = new T();
            obj.FirstName = firstName;
            obj.LastName = lastName;
            return obj;
        }
    }

    public interface IPerson
    {
        string FirstName { get; set; }
        string LastName { get; set; }
    }

    public class PersonDisplayer
    {
        private IPerson _person;

        public PersonDisplayer(IPerson person)
        {
            _person = person;
        }

        public string SimpleDisplay()
        {
            return String.Format("{1}, {0}", _person.FirstName,
_person.LastName);
        }

        public static string SimpleDisplay(IPerson person)
        {
            PersonDisplayer personDisplayer = new PersonDisplayer(person);
            return personDisplayer.SimpleDisplay();
        }
    }

    public class Customer : IPerson
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Company { get; set; }
    }

    public class Employee : IPerson
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int EmployeeNumber { get; set; }
    }
}
```

c# design-patterns solid-principles

Share
Improve this question
Follow

edited Oct 17, 2010 at 14:03
Péter Török
116k ● 31 ● 276 ● 331

asked Mar 18, 2009 at 17:11
Edward Tanguay
193k ● 319 ● 725 ● 1.1k

Similar question stackoverflow.com/questions/1215442/... – Michael Freidgeim Mar 12, 2013 at 20:49

5 Answers

Sorted by: Highest score (default)

8

I like to think of the Single Responsibility Principle as an implementation of separation of duties. Before I start splitting my classes as you have, I try to think of what each class should be responsible for.

Your classes are quite simple and lend themselves well to an abstract class with an implemented `Print()` and `Save()` functions as you mentioned. I would tend to keep that design over your current one.

However, if printing and saving were more complicated tasks which might be performed in different ways then a dedicated `Printer` or `Saver` class would be warranted, since that responsibility is now more complex. The 'complexity' threshold for making a new class is very subjective and will depend on the exact situation, but in the end, the code is just an abstraction for us lowly humans to understand, so make it such that it's the most intuitive.

You `Container` class is a little misleading. It doesn't actually 'contain' anything. It actually implements the Factory Method Pattern and would benefit from being named a factory.

Also, your `PersonDisplayer` is never instantiated and can provide all of its functionality through static methods, so why not make it a static class? It's not uncommon for utility classes such as Printers or savers to be static. Unless you have a need to have separate instances of a printer with different properties, keep it static.

Share Improve this answer Follow

answered Mar 18, 2009 at 17:39
Ben S
69.3k ● 31 ● 173 ● 214

PersonDisplayer is instatiated in its own static SimpleDisplay method – user76035 Mar 18, 2009 at 17:46

Right, that is a pattern I used a lot in PHP: a class with static methods which instantiate their own class then run a certain internal method on it, basically just giving the developer the ability to write one-liners instead of instantiating a class then calling a method on it.
– Edward Tanguay Mar 18, 2009 at 17:53

3 This is un-necessary object creation and will incur a performance penalty for no added functionality. The PersonDisplayer has no reason be be instantiated, all functionality can be provided statically. – Ben S Mar 18, 2009 at 18:34

4

I think you're on the right track. I'm not entirely sure about the Container class though. I'd generally stick with the simpler solution of just using "new" for these objects unless you have some business-driven need for that interface. (I don't consider "neat" to be a business requirement in this sense)

But the separation of "being" a customer responsibility from "displaying a customer" is nice. Stick with that, it's nice interpretation of SOLID principles.

Personally I have now *completely* stopped used any kind of static methods in this kind of code, and I rely on DI to get all the right service objects at the right place & time. Once you start elaborating further on the SOLID principles you'll find you're making a lot more classes. Try to work on those naming conventions to stay consistent.

Share Improve this answer Follow

answered Mar 18, 2009 at 17:28

krosenvold
77k ● 33 ● 156 ● 209

interesting, it just seems that my "customer" and "employee" classes are going to be very small, perhaps just properties and a little constructor, these types of classes used to be such cornerstones of the application – Edward Tanguay Mar 18, 2009 at 17:32

Yes, their responsibility is to "be" an employee. You may find that you'll be adding other methods too, not just properties. There's no reason to be religious about that. – krosenvold Mar 18, 2009 at 17:34

1

Well, I've never heard of this 'single responsibility principle' before, but what it appears to me that what you're doing by having these CustomerPrinter class and CustomerSaver classes is simply converting classes back to structs, and de-object-orienting everything.

For example, this would mean that different customer types would need different cases in the CustomerPrinter class if they needed to be printed differently. But as I understand it, one of the point of OO organisation, and of using inheritance trees and all that, is to do away with the need of this CustomerPrinter to know how to print everything: Customers know how to print themselves.

I don't believe in following these paradigms rigidly in any case. For example I'm unsure what the difference between an Interface and an Abstract Class is in your case. But then again I'm a C++ programmer not a C# programmer...

Share Improve this answer Follow

answered Mar 18, 2009 at 17:22

damian
3,664 ● 1 ● 29 ● 46

with an interface I can force my classes to have certain method signatures but I can't give them methods with functionality, for that I need an abstract class, I see interfaces as just a "light weight contract" that my classes need to be intelligently passed around the application – Edward Tanguay Mar 18, 2009 at 17:29

1

A few notes:

- Generally speaking SRP is all good, as is separation of display formatting from data.
- Considering display etc. I would rather think in terms of services, i.e. a PersonDisplayer is single, stateless and offers a string Display(IPerson) function. IMHO, a special class wrapper just to provide display does not provide any advantage.
- However, if you used data binding for wpf, you might have a DisplayablePerson class that would propagate PropertyChanged if Person changed. You would put DisplayablePerson objects into ObservableCollection and serve it as ItemsSource of some list control.
- What do you need Container for, is it only for instantiating and configuring instance?Try then Customer customer1 = new Customer{FirstName= "Jim", LastName= "Smith"};
- On a side note, I've tried object.Method < SomeType>(...) invocation a few times, as it seemed quickest and simplest solution. However, after some time I've always run into troubles with that one and ended up with object.Method(Type someTypeType, ...)

Share Improve this answer Follow

answered Mar 18, 2009 at 17:44

user76035
1,536 ● 1 ● 10 ● 12

0

You might have a look at IFormattable and IFormatProvider.

The framework has formatting classes for support.

Share Improve this answer Follow

answered Mar 18, 2009 at 18:15

shahkalpesh
33.5k ● 3 ● 65 ● 89