A Good Example of Liskov Substitution Principle

It's usually not easy to find good examples that explain the Liskov Substitution Principle (LSP) to developers. But really, how hard could it be? The definition is so simple:

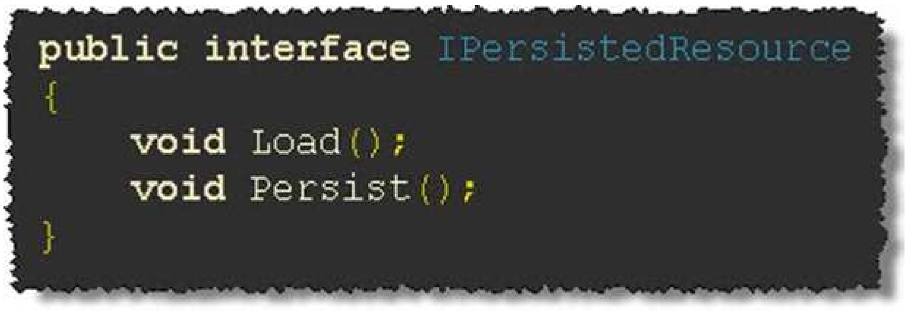
"What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."

Say what? I don't know about you, but that kind of definition usually flies way over my head. Some people use the *Rectangle versus Square* example to explain the principle, which is a good one, but that doesn't necessarily relate to things we normally do.

I like Uncle Bob's definition a lot better:

"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."

Recently we've run across a violation to that principle in a project. We have an interface defined like so:



```
public interface IPersistedResource
{
    void Load();
    void Persist();
}
```

It's a very small interface that represents resources that can be loaded in memory, and persisted afterwards in case there were changes to it.

Let's pretend we have the following implementations of that interface:

SEARCH

TWITTER

so, I guess it's time to abandon Netflix and go with Hulu? 29 minutes ago

What are the screen sharing options for the Mac? What are the alternatives to LiveMeeting (Windows-only) and GoToMeeting (too expensive)? 6 hours ago

Blogged: "RubyKoans: Great way to learn the Ruby language" http://bit.ly/qkrw0B 10 hours ago

Blogged: "Joshua's Bash presentation" http://bit.ly/pg9hh0 10 hours ago

hmm... simple and useful. I like it. RT @RookieOne: Visual Builds Board http://zite.to/roHPdj via @zite 11 hours ago

RECENT POSTS

- RubyKoans: Great way to learn the Ruby language
- Joshua's Bash presentation
- Do you control your source control? Or is it the other way around?
- Video from my SOLID talk at Agile.NET 2011 is up
- My Adventures in Rails Land

TAGS

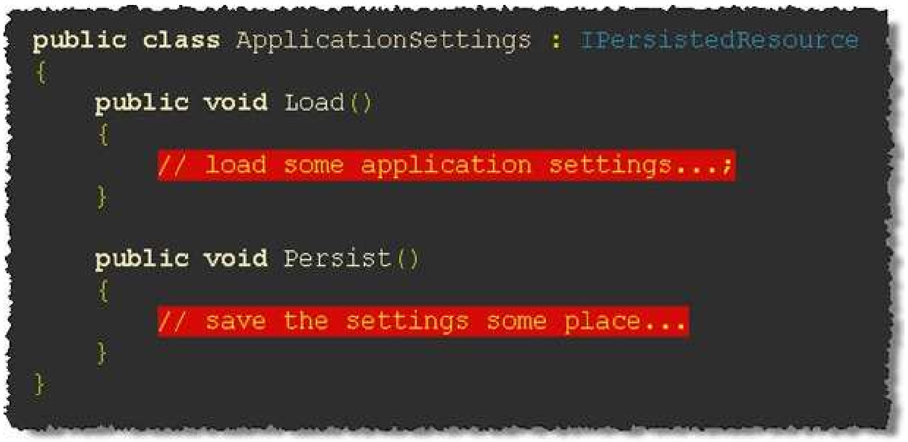
.net ASP.NET bdd book review books books code review books design patterns C# certification code comments CodeRush CodeRush ReSharper cucumber DDD CQRS User Experience distributed teams FoxPro FxCop FxCop C# VB houston techfest jigsaw puzzle LINQ linux mac mock rhino article mvc mvp mvvm patterns Pomodoro Techinique Evernote iPad productivity rails regular expression ruby screen capture shortcuts SOLID TDD tools testing tools User Experience virtual brown bag visual studio articles windows live writer WPF

CATEGORIES

Clean Code
CODE Announcements
Hobbies
Presentations
Productivity
Software Development
Uncategorized
Virtual Brown Bag

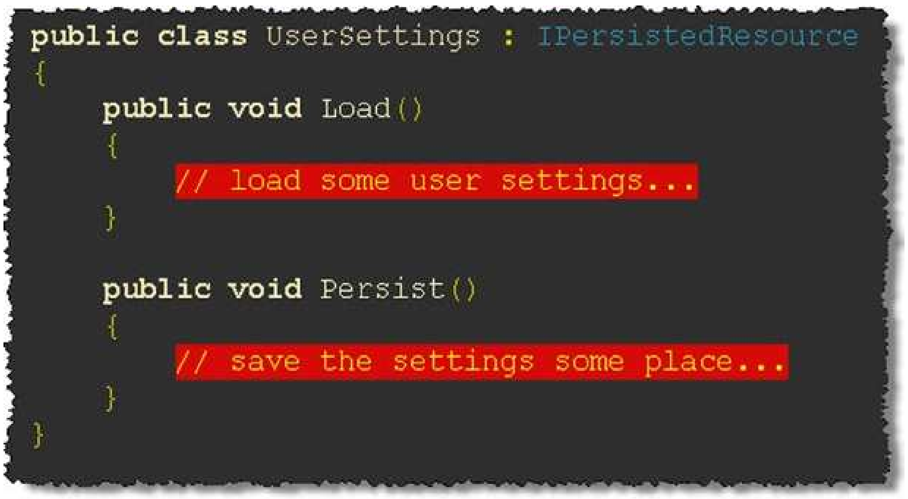
ARCHIVES

- July 2011
- June 2011
- May 2011
- April 2011
- February 2011



```
public class ApplicationSettings : IPersistedResource
{
    public void Load()
    {
        // load some application settings...;
    }

    public void Persist()
    {
        // save the settings some place...
    }
}
```

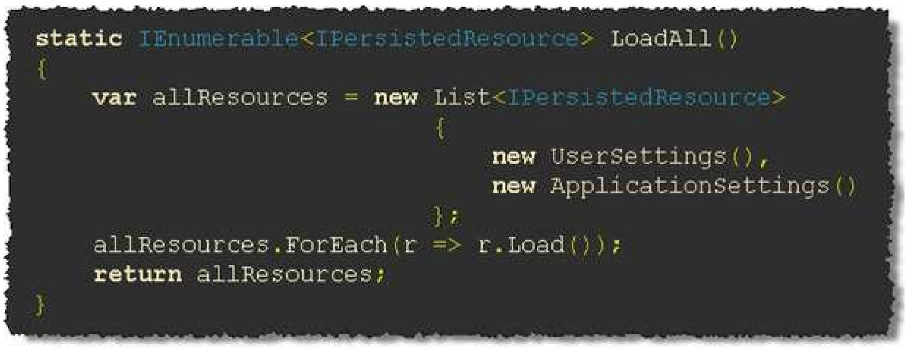


```
public class UserSettings : IPersistedResource
{
    public void Load()
    {
        // load some user settings...
    }

    public void Persist()
    {
        // save the settings some place...
    }
}
```

The actual implementation of those methods doesn't matter here; just assume that the real implementation loads and persists application and user settings.

Somewhere in the application we have some way to retrieve a list of instances of implementations of that interface, kind of like this:



```
static IEnumerable<IPersistedResource> LoadAll()
{
    var allResources = new List<IPersistedResource>
                        {
                            new UserSettings(),
                            new ApplicationSettings()
                        };
    allResources.ForEach(r => r.Load());
    return allResources;
}
```

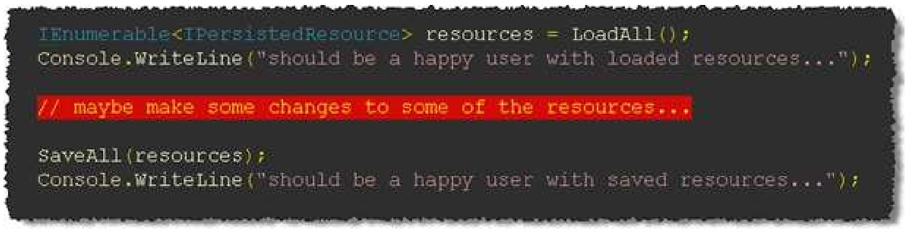
Some place else, we have a method that takes in a list of those objects, and call Persist on them:

And somewhere else we may use those methods, like so:

- January 2011
- December 2010
- November 2010
- October 2010
- September 2010
- August 2010
- May 2010
- April 2010
- March 2010
- February 2010
- January 2010
- December 2009
- November 2009
- October 2009
- September 2009
- August 2009
- July 2009
- May 2009
- April 2009
- March 2009
- February 2009
- January 2009
- December 2008
- November 2008
- October 2008
- September 2008
- August 2008
- July 2008
- June 2008
- May 2008
- April 2008
- March 2008
- February 2008
- January 2008
- December 2007
- November 2007
- October 2007
- September 2007
- August 2007
- July 2007
- June 2007
- May 2007
- April 2007
- March 2007
- February 2007
- January 2007
- December 2006
- October 2006
- August 2006
- April 2006
- March 2006
- October 2005
- August 2005

META

- Register
- Log in
- Entries RSS
- Comments RSS
- WordPress.com

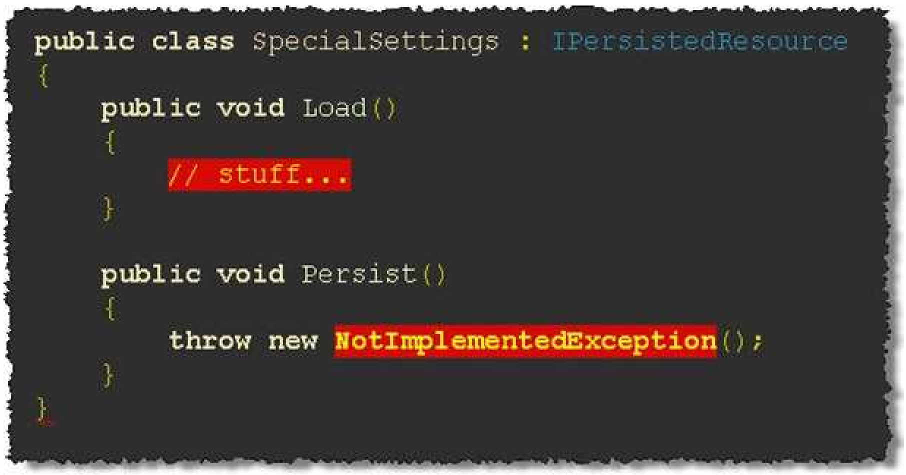


```
IEnumerable<IPersistedResource> resources = LoadAll();
Console.WriteLine("should be a happy user with loaded resources...");

// maybe make some changes to some of the resources...

SaveAll(resources);
Console.WriteLine("should be a happy user with saved resources...");
```

Everything works great, until a new class is added to the system in order to handle, let's say, some "special settings":

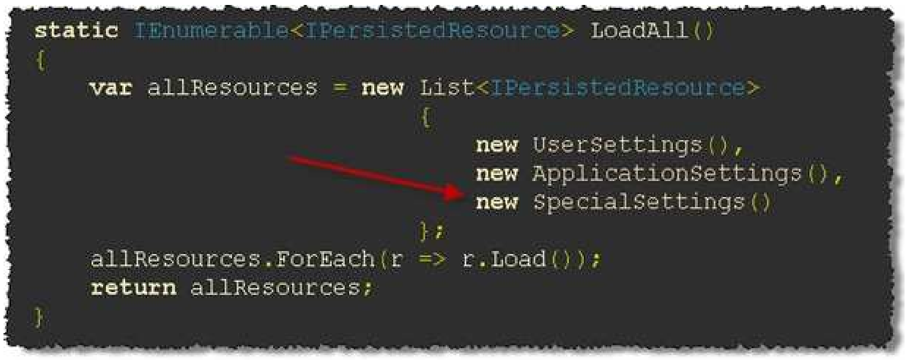


```
public class SpecialSettings : IPersistedResource
{
    public void Load()
    {
        // stuff...
    }

    public void Persist()
    {
        throw new NotImplementedException();
    }
}
```

It looks like the Load method does whatever stuff it's supposed to do in order to handle loading these special settings. The Persist method, on the other hand, throws a NotImplementedException. As it turns out, those settings are meant to be read-only, therefore, the Persist method can't really do anything.

The system is told to load the new class along with the other ones that implement that same interface:



```
static IEnumerable<IPersistedResource> LoadAll()
{
    var allResources = new List<IPersistedResource>
                        {
                            new UserSettings(),
                            new ApplicationSettings(),
                            new SpecialSettings()
                        };
    allResources.ForEach(r => r.Load());
    return allResources;
}
```

Now when we run the app everything should still work fine, until we hit the code that tries to persist all of those loaded resources, at which point we get a big and fat "NotImplementedException".

One (horrible) way to address this would be to change the SaveAll method:

If the specific resource being processed is of type SpecialSettings, we skip that one. Brilliant! Well, maybe not. Let's look back at a simplified definition of the Liskov Substitution Principle:

"An object should be substitutable by its base class (or interface)."

Looking at the SaveAll method it should be clear that "SpecialSettings" is NOT substitutable by its "IPersistedResource" interface; if we call Persist on it, the app blows up, so we need change the method to take that one problem into consideration. One could say "*well, let's change the Persist method on that class so it won't throw an exception anymore*". Hmm, having a method on a class that when called won't do what its name implies is just bad… really, really bad.

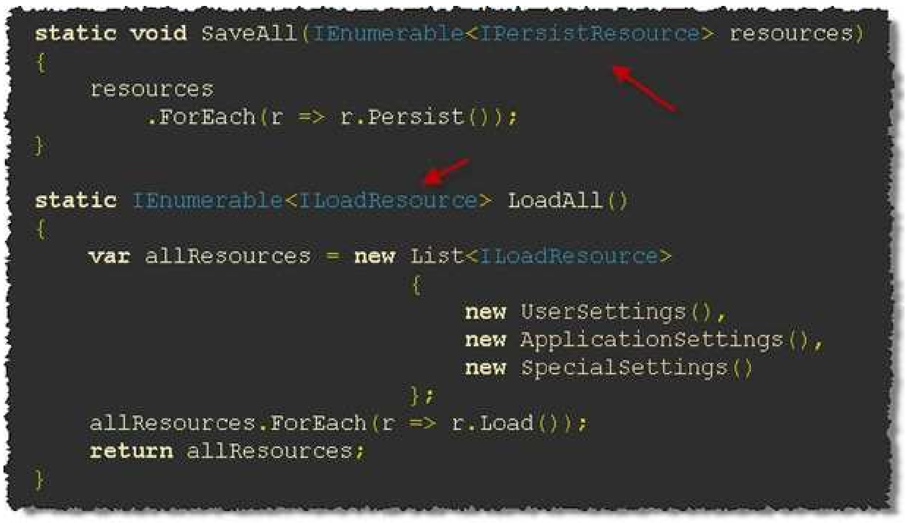
Write this down: anytime you see code that takes in some sort of baseclass or interface and then performs a check such as "if (someObject is SomeType)", there's a very good chance that that's an LSP violation. I've done that, and I know so have you, let's be honest.

Another great definition for LSP comes from this motivational poster that the folks at Los Techies put together:

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

So what's the fix?

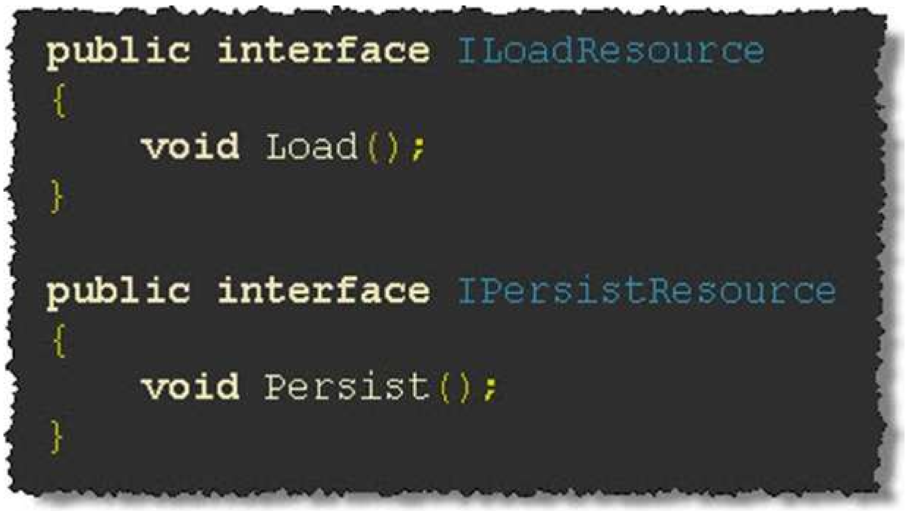
The fix here is to tailor the interface based on what each client needs (Interface Segregation Principle, or ISP). The LoadAll method (which is one client of those classes) is really only concerned about the "Load" capability, whereas the "SaveAll" method (another client) is only concerned about the "Persist" capability. In other words, these is what those clients need:



```
static void SaveAll(IEnumerable<IPersistResource> resources)
{
    resources
        .ForEach(r => r.Persist());
}

static IEnumerable<ILoadResource> LoadAll()
{
    var allResources = new List<ILoadResource>
                           {
                               new UserSettings(),
                               new ApplicationSettings(),
                               new SpecialSettings()
                           };
    allResources.ForEach(r => r.Load());
    return allResources;
}
```

The SaveAll takes in something tailored to its needs, *IPersistResource*'s, and the same goes for LoadAll, which only cares about *ILoadResource*'s (in the real app, the actual instantiation of these classes happen somewhere else). This is what the granular new interfaces look like:



```
public interface ILoadResource
{
    void Load();
}

public interface IPersistResource
{
    void Persist();
}
```

Yup, it's pretty much the former "IPersistedResource" split up into two separate interfaces, tailored to their client needs. Both the UserSettings and ApplicationSettings classes can implement these two interfaces, whereas the SpecialSettings class would only implement *ILoadResource*; this way, it isn't forced to implement interface members it can't handle.

Very often people ask what's the most appropriate number of members in an interface. In the real world example I gave here, the original interface had only 2 members; one could say that was small enough, but as it turns out, it wasn't. The IPersistedResource interface was doing too much (both loading *and* persisting stuff) based on the clients that use its implementers. In the end, two interfaces with a single method on them fit the bill a lot better. *Interfaces with single responsibility*? Yup, Single Responsibility Principle (SRP); as with design patterns, sometimes SOLID principles go hand in hand together.

DEDICATED POWER. CLOUD FLEXIBILITY. DEDICATED POWER. CLOUD FL
iWEB.COM
DEDICATED
SMART SERVER
ONLY AT iWEB.COM
FROM ONLY $85/MO
LEARN MORE
DEDICATED POWER. CLOUD FLEXIBILITY. DEDICATED POWER. CLOUD FLEXIBILITY. D

SOLID

This entry was posted on November 4, 2010, 10:42 am and is filed under Clean Code, Software Development, Virtual Brown Bag. You can follow any responses to this entry through RSS 2.0. You can leave a response, or trackback from your own site.

Like Be the first to like this post.

COMMENTS (2)

#1 by **Shinil** on January 5, 2011 - 11:06 am

A great post. Thank you very much

#2 by **Chan** on January 19, 2011 - 1:12 am

That is really great. I am clear now. Thanks!

Leave a Reply

Guest | Log In | Log In | Log In

Notify me of follow-up comments via email.

Notify me of new posts via email.