LSP: Liskov Substitution Principle

FUNCTIONS THAT USE ... REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.
http://www.objectmentor.com/resources/articles/lsp.pdf

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

Liskov Substitution Principle by Chad Myers

In my first (of hopefully more than one) post for The Los Techies Pablo's Topic of the Month - March: SOLID Principles effort, I'm going to talk about The Liskov Substitution Principle, as made popular by Robert 'Uncle Bob' Martin in The C++ Report.

I'm going to try as much as possible not to repeat everything that Uncle Bob said in the afore-linked PDF, you can go read the important stuff there. I'm going to try to give some real examples and relate this to the .NET world.

In case you're too lazy to read the link, let me start off with a quick summary of what LSP is: If you have a base class BASE and subclasses SUB1 and SUB2, the rest of your code should always refer to BASE and ***NOT*** SUB1 and SUB2.

A case study in LSP ignorance

The problems that LSP solves are almost always easily avoidable. There are some usual tell-tale signs that an LSP-violation is appearing in your code. Here's a scenario that walks through how an LSP-violation might occur. I'm sure we've all run into similar situations. Hopefully by walking through this, you can start getting used to spotting the trend up front and cutting it off before you paint yourself into a corner.

Let's say that somewhere in your data access code you had a nifty method through which all your DAO's/Entities passed and it did common things like setting the CreatedDate/UpdatedDate, etc.

```
public void SaveEntity(IEntity entity)
{
   DateTime saveDate = DateTime.Now;

   if( entity.IsNew )
      entity.CreatedDate = saveDate;

   entity.UpdatedDate = saveDate;

   DBConnection.Save(entity);
}
```

Clever, works like a champ. Many of you will hopefully have cringed at this code. I had a hard time writing it, but it's for illustration. There's a lot of code out there written like this. If you didn't cringe and you don't see what's wrong with that code, please continue reading. Now, the stakeholders come to you with a feature request:

Whenever a user saves a Widget, we need to generate a Widget Audit record in the database for tracking later.

You might be tempted to add it to your handy-dandy SaveEntity routine through which all entities pass:

```
public void SaveEntity(IEntity entity)
{
   WidgetEntity widget = entity as WidgetEntity;
   if( widget != null )
   {
      GenerateWidgetAuditEntry(widget);
   }

   // ...
```

Great! That also works like a champ. But a few weeks later, they come to you with a list of 6 other entities that need similar auditing features. So you plug in those 6 entities. A few weeks later, the come to you and ask you something like this:

When an Approval record is saved, we need to verify that the Approval is of the correct level. If it's not the correct level, we need to prompt the user for an excuse, otherwise they can't continue saving.

Oh boy, that's tricky. Well, now our SaveEntity looks something like this:

```
public void SaveEntity(IEntity entity)
{
   if( (entity as WidgetEntity) != null ){
      GenerateWidgetAuditEntry((WidgetEntity) entity);
   }

   if ((entity as ChocolateEntity) != null){
      GenerateChocolateAuditEntry((ChocolateEntity)entity);
   }

   // ...

   ApprovalEntity approval = entity as ApprovalEntity;
   if( approval != null && approval.Level < 2 ){
      throw new RequiresApprovalException(approval);
   }

   // ...
```

Pretty soon your small, clever SaveEntity method is 1,500 lines long and knows everything about every entity in the entire system.

Where'd we go wrong?

Well, there are several places to start. Centralizing the saving of entities isn't the greatest idea. Putting the logic for auditing whether entries need to be created or not into the SaveEntity method was definitely the wrong thing to do. Finally, due to the complexities of handling wildly differing business logic for different entities, you have a control flow problem with the approval level that requires the use of a thrown exception to break out of the flow (which is akin to a 'goto' statement in days of yore).

The concerns of auditing, setting created/updated dates, and approval levels are separate and orthogonal from each other and shouldn't be seen together, hanging around in the same method, generally making a mess of things.

More to the point of this blog post; SaveEntity violates the Liskov Substitution Principle. That is to say, SaveEntity takes an IEntity interface/base class but deals with specific sub-classes and implementations of IEntity. This violates a fundamental rule of object-oriented design (polymorphism) since SaveEntity pretends to work with any particular IEntity implementation when, in fact, it doesn't. More precisely, it doesn't treat all IEntity's exactly the same, some get more attention than others.

Why is this a problem? What if you were reusing your terribly clever SaveEntity method on another project and have dozens of IEntity implementations and the stakeholders for that project also wanted the auditing feature. Now you've got a problem.

Solutions

One fine approach to this problem of doing things at-the-moment-of-saving would be to use the Visitor Pattern as described by Matthew Cory in this post. Though, I would say in this particular example, there is a much deep-rooted and systemic design problem which revolves around the centralization of data access.

Another, in our case more preferable, way to go might be to use the repository pattern for managing data access. Rather than having "One Method to Rule them All", you could have your repositories worry about the Created/Updated date time and devise a system whereby all the repository implementations share some of the Created/Updated date entity save/setup logic.

As specific one-off problems arise (such as auditing, extra approval/verification, etc.), they can be handled in a similarly one-off manner. This is achieved by the individual entity's related repository (who knows all about that one type of entity and that's it). If you notice that several entities are doing the same sort of thing (i.e. auditing). You can, create a class and method to handle auditing in a common manner and provide the various repositories which need auditing with that functionality. Resist, if at all possible, the urge to create an 'AuditingRepositoryBase' class that provides the auditing functionality. Inevitably, one of those audit-requiring entities will have another, orthogonal concern for which you will have another *Base class and, since you can't do multiple inheritance in .NET, you are now stuck. Prefer composition of functionality over inheritance of functionality whenever possible.

If you have a rich domain model, perhaps the most elegant approach of all would be to make things like auditing a first-class feature of the domain model. Every Widget always has at least one WidgetAuditEntry associated with it and this association is managed through the domain logic itself. Likewise, the approval level would be best handled higher up in the logic chain to prevent last minute "gotchas" in the lifecycle that would require something less than elegant like an exception as a thinly veiled 'goto' bailout.