

Re: Proper design of classes

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2008-07/msg02659.html>

- *From:* "Leon Jollans" <myforename@xxxxxxxxxxxxxx>
 - *Date:* Thu, 24 Jul 2008 02:03:38 +0100
-

what you might end up having is a class explosion. I will suggest you use interface that inherits from other interfaces. Like you can have IPrivateCustomer, IBusinessCustomer, ICustomer. Your ICustomer Inherits from the IPrivateCustomer and IBusinessCustomer.

I disagree with this. Surely you're just replacing "class explosion" with "interface explosion" and without a business case for introducing interfaces at all. Interfaces are very powerful as we know for a lot of scenarios, but it hasn't been asked for here. All that's important is modelling the business domain appropriately in such a way as it can be programmed against.

now back to the OP

1) Lets say that I have this Customer class like I said, and I want to distinguish between different types of customers, for example private, business, other etc. This will allow me to filter customers based on their type. Should I derive from a base class Customer or should I make it a property of the class, perhaps an Enum with the relevant options.

There are of course lots of approaches and possible answers, all of which depend on what you're trying to achieve rather than there being any *right* way of designing it. Sure, you could go for a property on the Customer describing its type. It's easy and effective. I've done things like this before quite successfully, however that's in relatively simple applications. One thing I would say if you go this way is don't use an enum. An enum won't leave you any room for adding new Customer types without recompilation of the full application – and even when you do that it's entirely possible that the new enum value will cause client code to silently fail by not recognising the new customer type. Far better to have a CustomerType class (perhaps even loaded from your DB)

In more general terms, personally, I'd be more inclined to store common data in a base Customer class, and move the functionality to subclasses or external code where possible.

Very often people put every last bit of functionality in a single object, and I think this is bad design. You could use the customer instance to obtain extended information, or put specific data in an associated set of CustomerInfo classes.. This is a bit like the interface approach offered earlier, but it doesn't require you to implement each interface in your customer class. For example

```
class CustomerInfo;
```

Re: Proper design of classes

```
class CompanyInfo;
class ProspectInfo;
class IntranetUserCustomerInfo
```

```
Customer customer = LoadCustomer("cust-id");
ProspectInfo prospect = PropsectInfo.LoadFor(customer);
if( prospect.LikelihoodOfClosure < 0.5 ) { SendMarketingInfo(prospect); }
```

etc..

I prefer the functionality-driven approach. So you have your base customer data, in a base class which is good enough for a lot of scenarios. but then you instantiate specific Customer subclasses depending on the context, each that imply differnt behaviour. However – and this is the key thing – any of them can be instatiated against the same set of customer data (eg a database record ID). Loading the core customer data can be then done either by the base class alone, or by any subclass (and hence the base class too), and any other behaviour can be handled by the subclass – if used.

So this example describes a task-based hierarchy which provides a base customer that does all the common stuff, loading name, email address etc, but it allows implementation of a task list for any number of contextual subclasses... and this can be extended by third party code .. whatever.

// this is the base class. All Customers in the task based system, regardless of the context can be treated as Customers.

```
public class Customer
{
// this method will call LoadSpecificData()
// which does nothing here, but it can be overridden by
// subclasses to obtain any further info required
public void Load(int recordID)
{
Data data = GetDataFromDB(recordID); // DataSet, XmlNode etc
AssignCommonCustomerDataToFields(data, recordID);
LoadSpecificData(data);
}
}
```

```
// override in subclasses to load context specific data.
// The same model can, and should be used for saving records too.
protected virtual void LoadSpecificData(Data data, int recordID)
{
// NOOP
}
}
```

```
// this subclass defines a new method that must be implemented.
// The WorkItem class is implied, and expected to be subclassed too.
public abstract class TaskBasedCustomer : Customer
{
// example specialisation
public abstract IEnumerable<WorkItem> GetWorkItems();
{ yield break; }
}
```

Re: Proper design of classes

```
// this describes the email communication context of the customer
public class CommunicatingCustomer : TaskBasedCustomer
{
// .. specific properties..
```

```
// from Customer
override void LoadSpecificData(Data data, int recordID)
{
LoadContactInformation(data);
}
```

```
// from TaskBasedCustomer
override IEnumerable<WorkItem> GetWorkItems()
{
foreach(Email received in GetInbox())
{
yield return new EmailReceivedWorkItem(received);
}
}
}
```

```
// this describes the helpdesk/support context of the customer
public class HelpdeskCustomer : TaskBasedCustomer
{
// .. specific properties..
```

```
// from Customer
override LoadSpecificData(Data data, int recordID)
{
LoadIncidentHistoryFromCRM(recordID);
}
```

```
// from TaskBasedCustomer
override IEnumerable<WorkItem> GetWorkItems()
{
foreach(Incident logged in IncidentHistory)
{
if(logged.Status == IncidentStatus.Open)
{
yield return new OpenIncidentWorkItem(logged);
}
}
}
}
```

now what does this give you... well, in this example, you could have a master list of customers; with customer ids, and this could generate a link to an Inbox page like `inbox?id=[customerId]`. and that page's codebehind looks like this

Re: Proper design of classes

```
TaskBasedCustomer customer = new CommunicatingCustomer();
customer.Load(Request["customerId"])
InboxDataGrid.DataSource = customer.GetWorkItems();
InboxDataGrid.DataBind();
```

or you could have a helpdesk info page listing the customers incidents that works like this:

```
TaskBasedCustomer customer = new HelpdeskCustomer();
customer.Load(Request["customerId"])
IncidentDataGrid.DataSource = customer.GetWorkItems();
IncidentDataGrid.DataBind();
```

note that each code block here is almost identical and so could be refactored further.. the only difference is the class you instantiate to view the customer detail. This way, you're never saying this customer *is a* Helpdesk customer, you're just saying they're a Customer, and the class hierarchy exists only to provide an appropriate API to the data depending on how you want to use it.

Of course this might be way off, and wildly inappropriate for what you want to achieve.

And so the short answer to your question is... It depends what you want to do. You may like some, all or none of these approaches. There's no *right* way to define a class hierarchy per se. I see a lot of advice given, articles, books, received wisdom on the subject etc that purports to have the *right* way of doing things; of designing class hierarchies.. but almost without exception (no pun intended) they're only right if the design suits your purpose.

2) ... for example PrivateCustomer adds PrivateName and LastName, and BusinessCustomer adds CompanyName (although I'm not sure that this is good design in term of good OO practice, where classes should be derived if they have different behaviour, not data).

I disagree with this too. I don't think it's bad OO practice to have a subclass which contains more specific data.

because in every document, for example Invoice, you'd see fields like "Company/Last Name: _____", which I don't like.

yeah – this sounds awful. Never do this. ;)

Leon

"Ahmed Salako" <Ahmed Salako@xxxxxxxxxxxxxxxxxxxxxxxxxxxx> wrote in message news:831B4E19-6559-4852-9BE3-265F3C8C6400@xxxxxxxxxxxxxxxxxxxx

Hiya,
what you might end up having is a class explosion. I will suggest you use interface that inherits from other interfaces. Like you can have

Re: Proper design of classes

Re: Proper design of classes

IPrivateCustomer, IBusinessCustomer, ICustomer. Your ICustomer Inherits from the IPrivateCustomer and IBusinessCustomer.

So you will have an implementation class called customer which implements ICustomer, since your ICustomer interface inherits from different interfaces, you will explicitly implement the other interfaces as well. So you can dynamically at runtime cast your Customer class to those different interface(s) when you need their certain properties, method or functionality. For example:

```
public interface IPrivateCustomer
{
    string PrivateName { get; set; }
    string PrivateAddress { get; set; }
}

public interface IBusinessCustomer
{
    string CompanyName { get; set; }
}

public interface ICustomer : IPrivateCustomer, IBusinessCustomer
{
    string FirstName { get; set; }
    string LastName { get; set; }
}

public class Customer : ICustomer
{
    //Explicitly implements all methods/properties of the three interfaces.
}
```

Create an instance of the Customer Class here by doing the any or all of following :

```
Customer customer = new Customer();
```

```
IPrivateCustomer privateCustomer = customer as IPrivateCustomer;
IBusinessCustomer businessCustomer = customer as IBusinessCustomer;
ICustomer iCustomer = customer as ICustomer;
```

The flexibility that you gain from this approach is that you only have a single customer class and you can add new interface and remove others.

Also you may want to look at the Factory pattern, which will support the creation of an object based on the type you supplied. You can use the combination of the factory pattern with the simple scenario that i ahve given.

Re: Proper design of classes

I hope this work.

Cheers and Enjoy.