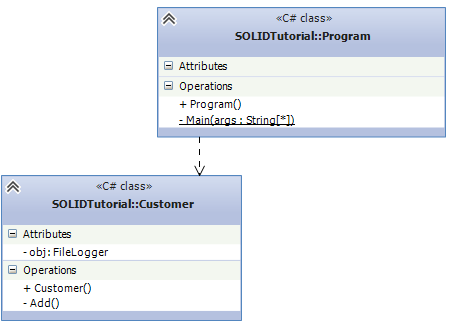
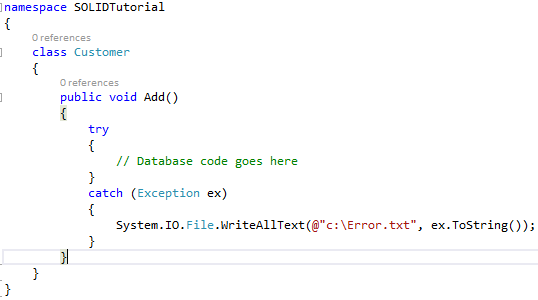
Supplementary Notes for SOLID architecture principles using simple C# examples - [Shivprasad koirala](http://www.codeproject.com/script/Membership/View.aspx?mid=1335831)

**S.O.L.I.D** is an acronym for the **first five object-oriented design(OOD) principles** by Robert C. Martin, popularly known as [Uncle Bob](http://en.wikipedia.org/wiki/Robert_Cecil_Martin).

Available at <http://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp>

**Issue 1:** The class below violates Single responsibility principle which states - A class should have one and only one reason to change, meaning that a class should have only one job.

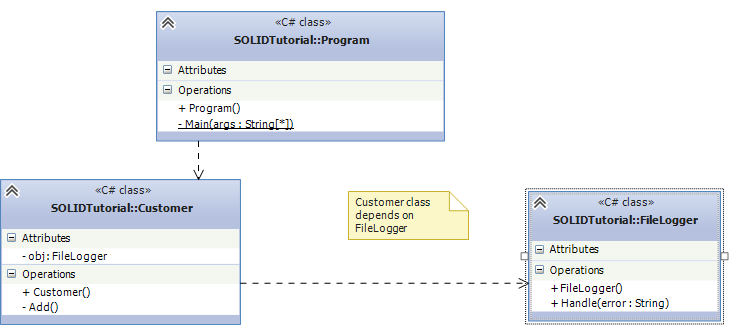
 

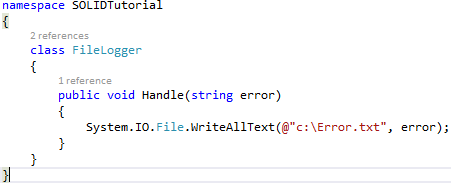
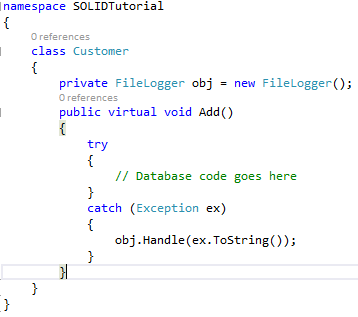
Customer class should do customer data validations, call the customer data access layer etc. but it also doing LOGGING activity. In simple words its over loaded with lot of responsibility. Also in the future if there is a need for a new logger like event viewer then there is a need to go and change the “Customer”class.

Logging is a cross cutting concern. Lots of applications even ones in different functional areas need to log errors and messages. So we need a consistent clever way to handle logging.

**Solution to Issue 1:**

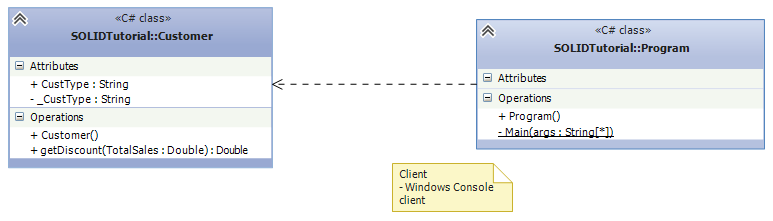
Applying SRP we can move that logging activity to some other class who will only look after logging activities.

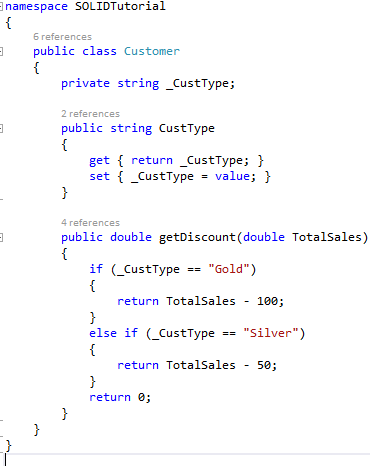
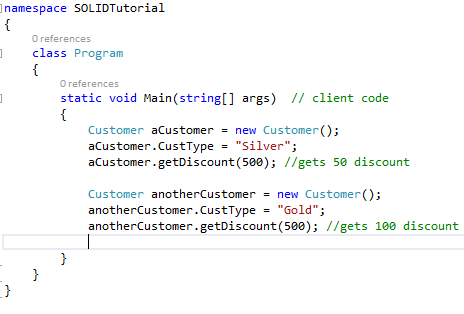


Now customer class has happily delegated the logging activity to the “FileLogger” class and customer class can concentrate on customer related activities.

**Issue 2:** Added a customerType attribute to Customer – violates Open Closed Principle which states - Objects or entities should be open for extension, but closed for modification.

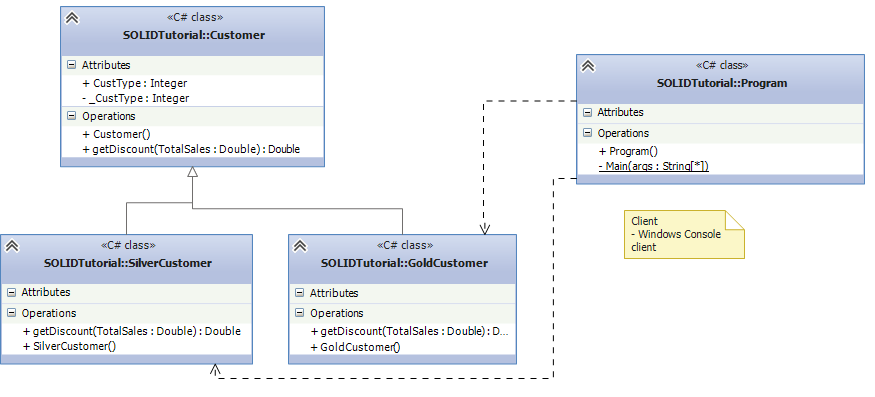


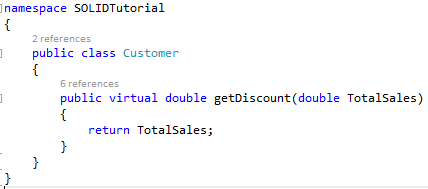
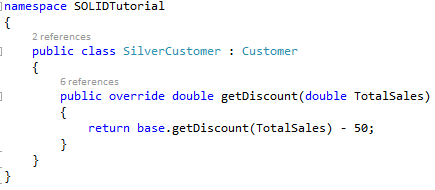
 

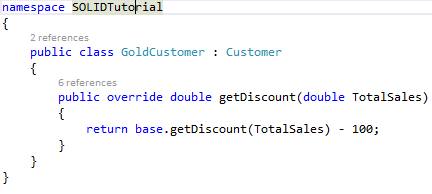
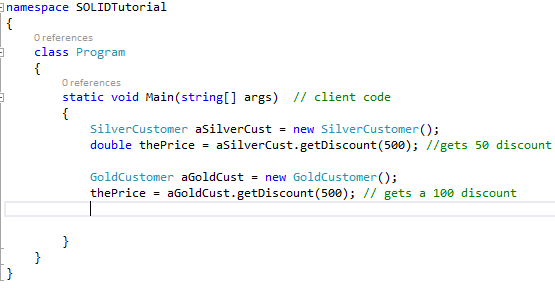
The problem is if we add a new customer type we need to go and add one more “IF” condition in the “getDiscount” function, in other words we need to change the customer class. Each different kind of customer receives different discounts.

**Solution to Issue 2:**

Rather than “MODIFYING” go for “EXTENSION”. In other words every time a new customer type needs to be added add a new subclass.

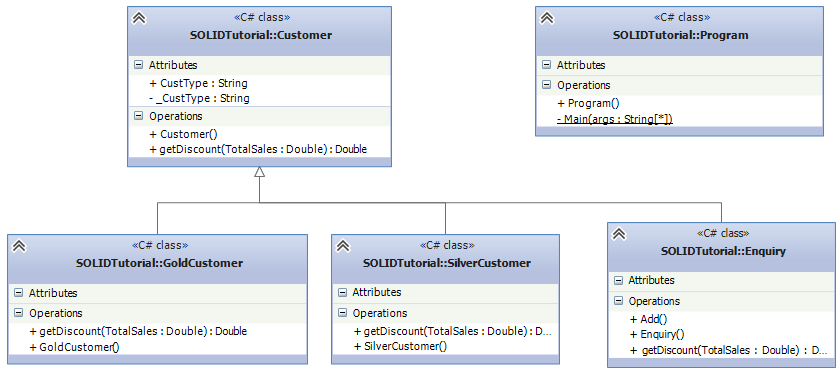


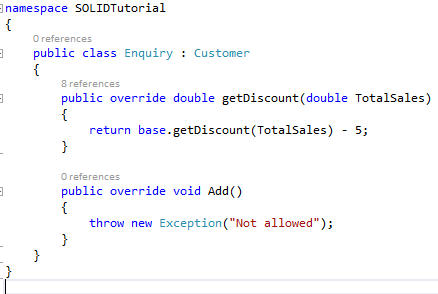
 

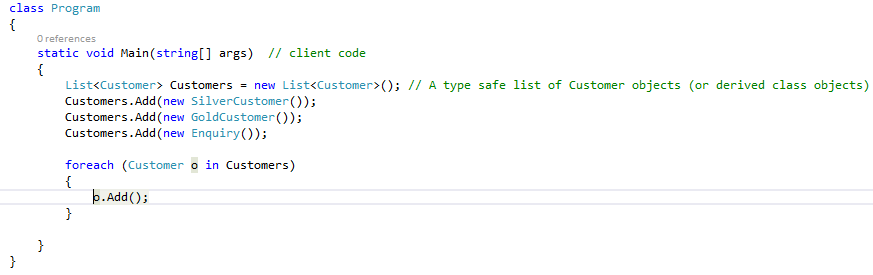
**Issue 3:** Added Enquiries and these are not actual customer’s they are just leads. Because they are just leads we do not want to save them to database for now.

So we create a new class called as Enquiry which inherits from the “Customer” class. We provide some discounts to the enquiry so that they can be converted to actual customers and we override the “Add’ method with an exception so that no one can add an Enquiry to the database. This violates Liskov substitution principle which states - All this is stating is that every subclass/derived class should be substitutable for their base/parent class.





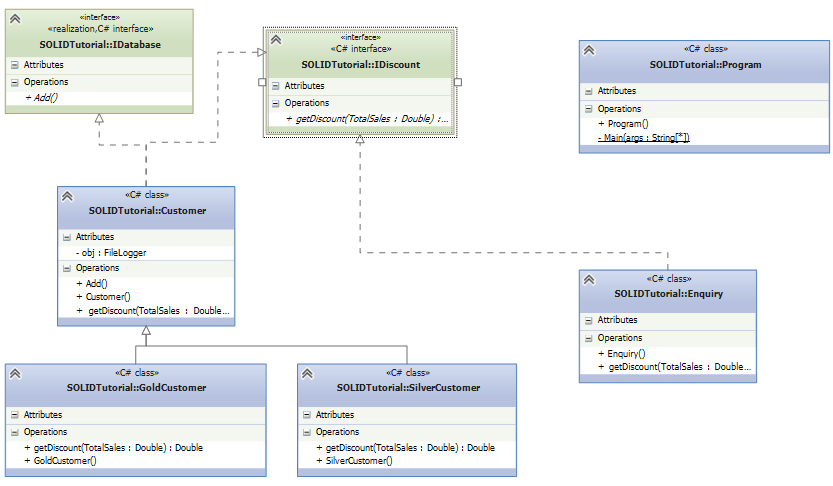
With any inheritance hierarchy the we should be able to use a superclass reference to refer to a subclass object and we do not expect any unusual behaviour. Below is client code which causes a problem at runtime.

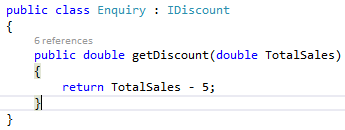
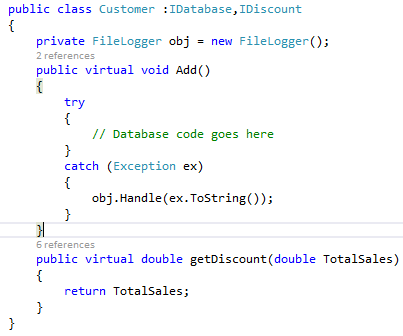


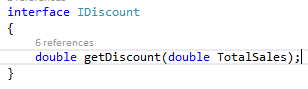
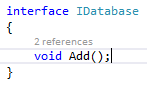
**Solution to Issue 3**

We should use an interface instead as Enquiries are not customers. They are not consistent with customers in every respect and so should not be part of the Customer inheritance hierarchy.

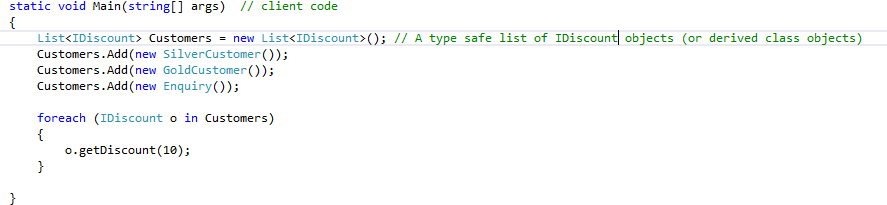
The diagram below shows the Enquiry implanting the IDiscount interface. The Customer class implements both IDatabase and IDiscount.



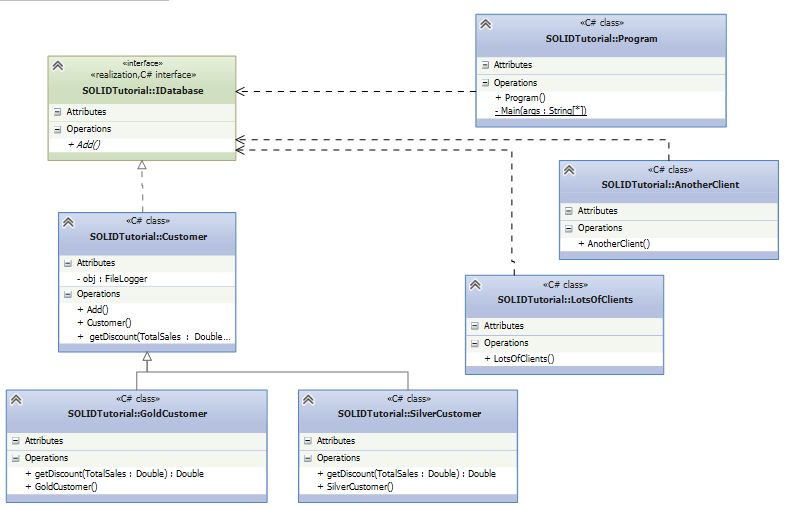




Now you can’t add an Enquiry to a Customer list. You can add an Enquiry to an IDiscount list.



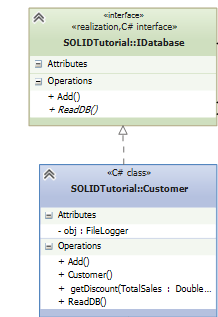
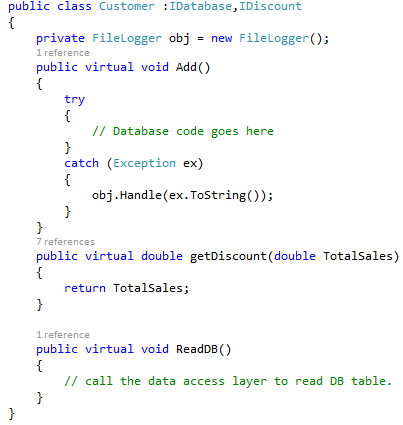
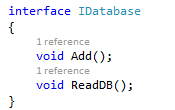
**Issue 4:** Suppose the customer class and it’s subclasses class are very successful and are utilised ‘reused’ by lots of clients as shown below.



Its known that in the future clients will require Add and another method called ReadDb(). In effect you have two kinds of client’s:

1. Who want’s just use “Add” method.
2. Others who wants to use “Add” + “ReadDB”.

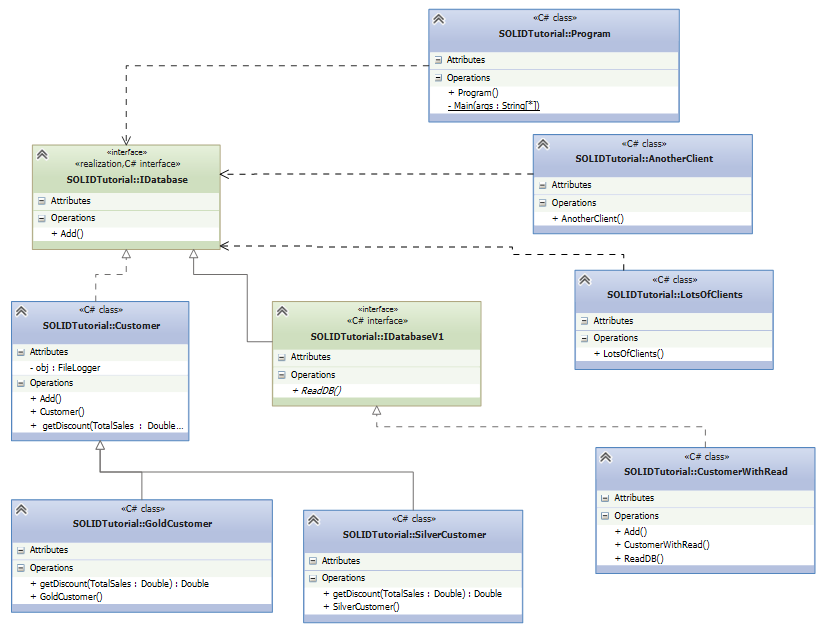
It’s tempting to just modify the IDatabase to include the ReadDB() method and add an implementation for it in the Customer class.

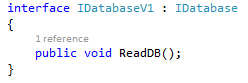
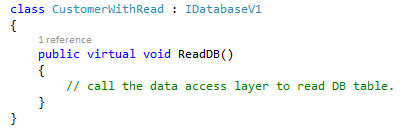
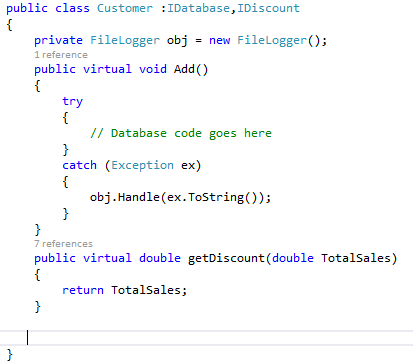
  

This breaks Interface Segregation Principle which states - A client should never be forced to implement an interface that it doesn’t use or clients shouldn’t be forced to depend on methods they do not use.

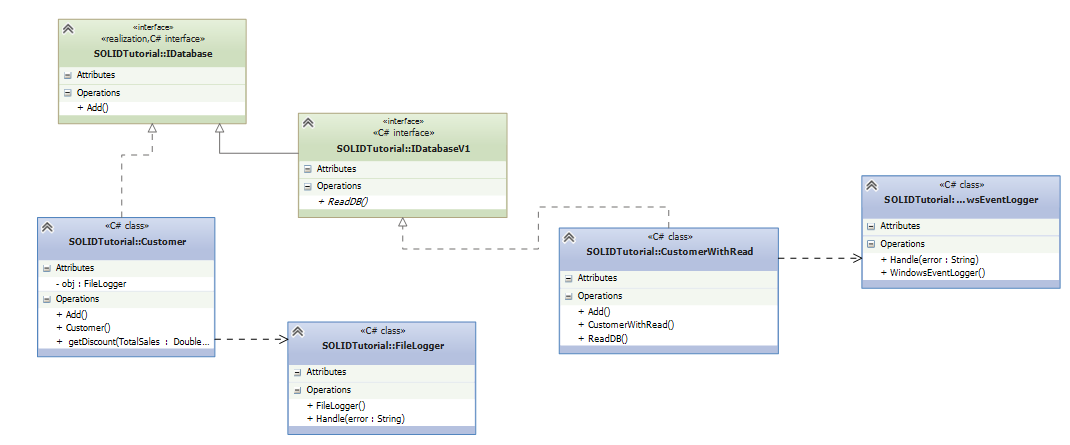
**Solution to Issue 4**

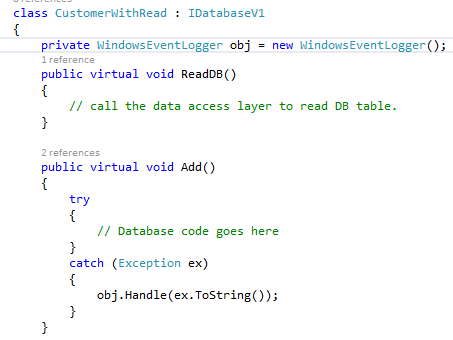
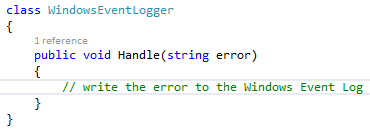
A much better idea is to leave the existing interface as is and derive a new interface from it. The existing clients continue to use the original interface and the new clients use the new interface which specifies the new functionality as per diagram below.



**Issue 5:** We created a FileLogger earlier, suppose there is a requirement for new types of loggers in the future starting with a WindowsEventLogger for the CustomerWithRead class.

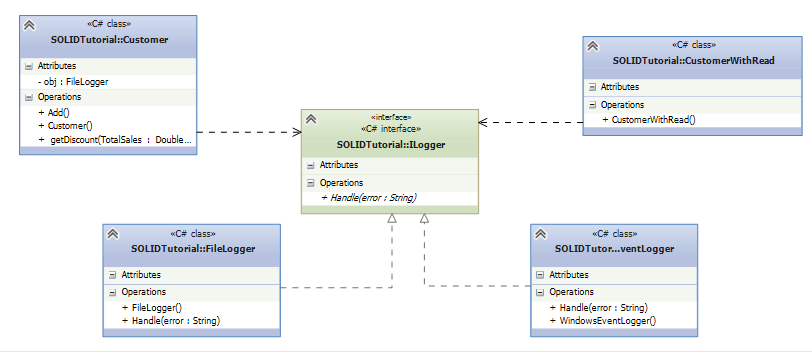


Problems arise when we have additional loggers or want to change the way a cross cutting concern such as logging is done as we have not adhered to the Dependency Inversion principle which states - Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

**Solution to Issue 5**

A much better approach is to create a common interface and using this common interface new logger types can be created.





The idea is to invert/delegate the responsibility for creating the Logger to someone else rather the customer class doing it. The code above utilises constructor injection to inject the dependency and the code below shows the client code.

