Übungsblatt 6 (Lösung)

Aufgabe 16 Das Single Responsibility Prinzip

Bei dieser Aufgabe die Verantwortung für Logging Aktivitäten ausgelagert werden. Eine mögliche Definition der neuen *FileLogger* Klasse könnte folgendermaßen aussehen:

```
class FileLogger : ILogger
{
    public void LogError(String error)
    {
        using (var writer = File.AppendText("error.log"))
        {
            writer.WriteLine(error);
        }
    }
}
```

Das Interface *ILogger* wird später definiert. Die Klasse *Customer* sieht daher zu diesem Zeitpunkt folgendermaßen aus:

```
class Customer : IDatabase
{
    Int32 customerType;
    ILogger logger;

    public Customer(ILogger logger)
    {
        this.logger = logger;
    }

    public ILogger Logger
    {
        get { return logger; }
    }

    public Int32 CustomerType
    {
        get { return customerType; }
        set { customerType = value; }
    }

    public virtual void Add()
    {
        try
        {
            //Add to database
            //throw new Exception("Test exception!!");
        }
        catch (Exception ex)
```

```
        {
            logger.LogError(ex.Message);
        }
    }

    /* ... */
}
```

Im folgenden soll die Klasse *Customer* noch flexibler gestaltet werden.

Aufgabe 17 Das Open Close Prinzip

Mit dem Open Close Prinzip soll eine Designrichtlinie eingehalten werden, die stark auf Polymorphie setzt. Zunächst sollte daher die Klasse *Customer* abstrakt gesetzt werden.

```
abstract class Customer : IDatabase
{
    /* ... */

    public abstract Double GetTotalDiscount(Double totalSales);
}
```

Anschließend sollen neue Konkretisierungen erstellt werden. Beispielsweise ist der Code für den *GoldCustomer* folgendermaßen aufgebaut:

```
class GoldCustomer : Customer
{
    public GoldCustomer(ILogger logger) : base(logger)
    {
    }

    public override Double GetTotalDiscount(Double totalSales)
    {
        return totalSales - 100.0;
    }
}
```

Letztlich ist daher keine Änderung mehr an bestehenden Methoden notwendig. Wenn es einen neuen Typ von Kunden gibt, muss die *GetTotalDiscount* Methode implementiert werden.

Aufgabe 18 Das Liskov Substitution Prinzip

Die Gestaltung der Klasse *Enquiry* ist nicht sehr geglückt. Zunächst wird ein Interface erstellt, welches die gemeinsamen Fähigkeiten von *Customer* und *Enquiry* ausdrückt.

```
interface IDiscount
{
    Double GetTotalDiscount(Double totalSales);
}
```

Anschließend kann die Klassen optimiert werden. Die nicht mehr notwendige Methode *Add* wird dabei vollständig gelöscht, da diese nur zu Fehlern geführt hätte.

```
class Enquiry : IDiscount
{
```

```
    IDiscount customer;

    public Enquiry(IDiscount customer)
    {
        this.customer = customer;
    }

    public Double GetTotalDiscount(Double totalSales)
    {
        return customer.GetTotalDiscount(totalSales) - 5;
    }
}
```

Die Berechnung der *GetTotalDiscount* Methode baut auf der Berechnung einer anderen *IDiscount* Instanz auf. Dies wird i.d.R. ein *Customer* Objekt sein.

Aufgabe 19 Das Interface Segregation Prinzip

Das ISP ist in der Definition von *IDatabase* verletzt. Die einfachste Art und Weise ISP einzuhalten besteht darin, einfach eine Operation pro Schnittstelle zu definieren. In diesem Fall handelt es sich somit um zwei Interfaces:

```
interface IValidate
{
    Boolean Validate();
}
interface IDatabase
{
    void Add();
}
```

Die Klasse *Customer* ändert sich hierbei kaum, d.h. sie wird nicht um die Implementierung eines weiteren Interfaces ergänzt.

```
abstract class Customer : IDatabase, IDiscount
{
  /* ... */
}
```

Stattdessen ist es möglich eine weitere Pseudo-*Customer* Klasse zu gestalten. Diese Klasse implementiert zusätzlich *IValidate*. Im Prinzip handelt es sich hierbei um einen Wrapper für *Customer*.

```
class ValidateCustomer : Customer, IValidate
{
    Customer customer;

    public ValidateCustomer(Customer customer) : base(customer.Logger)
    {
        this.customer = customer;
    }

    public override Double GetTotalDiscount(Double totalSales)
    {
        return customer.GetTotalDiscount(totalSales);
    }
```

```
    public Boolean Validate()
    {
        //Some validation code
        return true;
    }
}
```

Diese Klasse kann nun genutzt werden um beliebige *Customer* zu validieren.

Aufgabe 20 Das Dependency Inversion Prinzip

In dieser Lösung wurde das DIP bereits von Anfang an eingesetzt. So stellt der Konstruktor der *Customer* Klasse bereits die Anforderung, dass die Abhängigkeit *ILogger* aufgelöst werden muss. Dies macht die Erstellung einer *ILogger* Instanz **vor** der Erstellung eines *Customer* Objektes notwendig.
Das *ILogger* Interface sieht folgendermaßen aus:

```
interface ILogger
{
    void LogError(String error);
}
```

Eine weitere beispielhafte Implementierung kann in Form einer *EventLogger* Klasse geschehen.

```
class EventLogger : ILogger
{
    public void LogError(String error)
    {
        EventLog.WriteEntry("SOLID Application", error);
    }
}
```

Abschließend noch ein Blick auf das vollständige Projekt, welches nun alle Anforderungen der SOLID Prinzipien erfüllt.