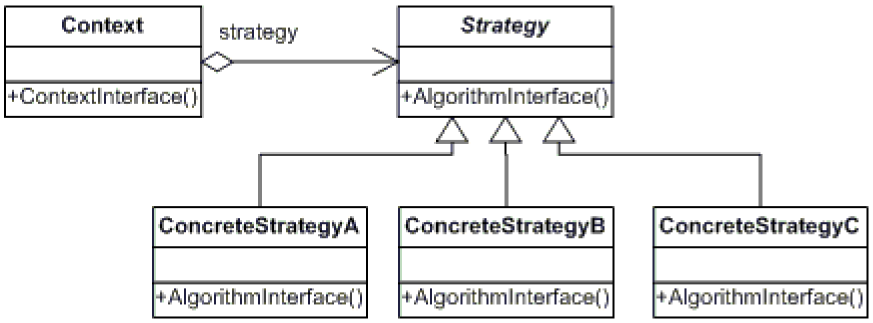
CPSC 501 – Assignment #8
Strategy and Template Design Patterns

Strategy Pattern:

Strategy pattern allows encapsulation of related algorithms so that one algorithm can be easily interchanged with another one.



Create a new windows forms application project called StrategyPattern. Add an interface called IStrategySort with the code as shown below. This interface defines the contract that is required by all algorithm classes that can be provided as part of the strategy. The interface requires that the type T should further implement IComparable interface so that we can sort objects of this type T.

```
interface IStrategySort<T>
    where T: IComparable, new() // T should implement IComparable and
{                               // T should provide a constructor
    void DoSort(List<T> TList);
}
```

Add a class called Student to the project with the following code in it.

```
class Student : IComparable
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Id { get; set; }
    public int Test1Score { get; set; }
    public int Test2Score { get; set; }

    public int CompareTo(object obj)
    {
        int ret = 0;
        if (obj is Student)
        {
            ret = this.LastName.CompareTo(((Student)obj).LastName);
        }
        return ret;
    }

    public override string ToString()
    {
        return FirstName + " " + LastName + " : " + Id.ToString() +
            " : " + Test1Score.ToString() + " : " + Test2Score.ToString();
    }
}
```

Create a concrete strategy class called ShellSortStrategy with the code as shown below.

```
    class ShellSortStrategy : IStrategySort<Student>
    {

        public void DoSort(List<Student> TList)
        {   // Shell Sort
            int i, j, increment;
            Student temp;
            increment = 3;
            while (increment > 0)
            {
                for (i = 0; i < TList.Count; i++)
                {
                    j = i;
                    temp = TList[i];

                    while ((j >= increment) && (TList[j - 
increment].CompareTo(temp)>0))
                    {
                        TList[j] = TList[j - increment];
                        j = j - increment;
                    }

                    TList[j] = temp;
                }

                if (increment / 2 != 0)
                {
                    increment = increment / 2;
                }
                else if (increment == 1)
                {
                    increment = 0;
                }
                else
                {
                    increment = 1;
                }
            }
        }
    }
```

Add another concrete strategy class called QuickSortStrategy with the code as shown below.

```
    class QuickSortStrategy : IStrategySort<Student>
    {
        public void DoSort(List<Student> TList)
        {
            TList.Sort();  // Quick sort from .Net library
        }
    }
```

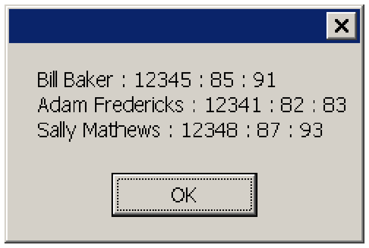
Add a SortContext class to the project that acts as an abstraction for the different algorithms that implement IStrategySort as shown below.

```
class SortContext  // abstraction for Strategy or Algorithm classes
{
    IStrategySort<Student> istrat = null;

    public SortContext(IStrategySort<Student> ist)
    {
        istrat = ist;
    }

    // algorithm interface
    public void DoSort(List<Student> TList)
    {
        istrat.DoSort(TList);
    }
}
```

To test the strategy pattern, add a button to the form called btnStrategySort with the following code in it.

```
private void btnStrategySort_Click(object sender, EventArgs e)
 {
     List<Student> STList = new List<Student>();
     Student s1 = new Student
     {
         FirstName = "Bill",
         LastName = "Baker",
         Id = 12345,
         Test1Score = 85,
         Test2Score = 91
     };
     STList.Add(s1);

     Student s2 = new Student
     {
         FirstName = "Sally",
         LastName = "Mathews",
         Id = 12348,
         Test1Score = 87,
         Test2Score = 93
     };
     STList.Add(s2);

     Student s3 = new Student
     {
         FirstName = "Adam",
         LastName = "Fredericks",
         Id = 12341,
         Test1Score = 82,
         Test2Score = 83
     };
     STList.Add(s3);

     SortContext cxt = new SortContext(new ShellSortStrategy());
     cxt.DoSort(STList);
     string out1 = "";
     foreach (Student st in STList)
         out1 += st.ToString() + "\n";
     MessageBox.Show(out1);

     // switch to Quicksort
     SortContext cxt2 = new SortContext(new QuickSortStrategy());
     cxt.DoSort(STList);
     string out2 = "";
```

```
            foreach (Student st in STList)
                out2 += st.ToString() + "\n";
            MessageBox.Show(out2);
        }
```



```
Bill Baker : 12345 : 85 : 91
Adam Fredericks : 12341 : 82 : 83
Sally Mathews : 12348 : 87 : 93
```

To understand the context class a little, better, in a real application, it could be, for example a University class that maintains a list of Students as shown below.

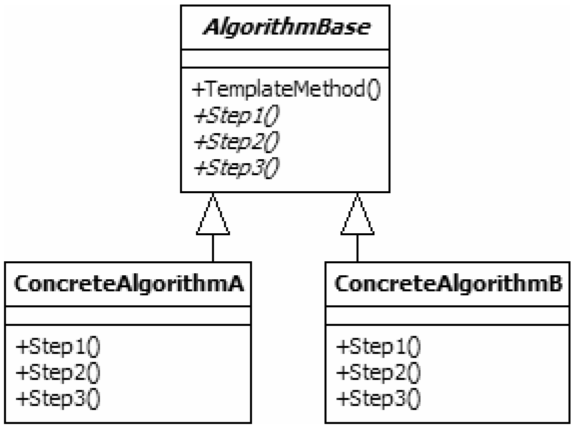
```
    class University  // Context class that maintains a list of Students
     {
        private IStrategySort<Student> _sortStrategy;
        internal IStrategySort<Student> SortStrategy
        {
            get { return _sortStrategy; }
            set { _sortStrategy = value; }
        }

        public University(IStrategySort<Student> sortStrategy)
        {
            this._sortStrategy = sortStrategy;
        }

        private List<Student> _STList = new List<Student>();
        public List<Student> STList
        {
            get { return _STList; }
            set { _STList = value; }
        }

        public void AddStudent(Student st)
        {
            _STList.Add(st);
        }

        public void SortStudent()
        {
            _sortStrategy.DoSort(STList); // will sort via Shell or QuickSort
        }
    }
```

Now our code to test the strategy pattern via the University context class will appear as:

```
private void btnStrategySort_Click(object sender, EventArgs e)
        {
            IStrategySort<Student> ist = new ShellSortStrategy();
            University u1 = new University(ist);
            Student s1 = new Student
            {
                FirstName = "Bill",
                LastName = "Baker",
                Id = 12345,
                Test1Score = 85,
                Test2Score = 91
            };
            u1.AddStudent(s1);

            Student s2 = new Student
            {
                FirstName = "Sally",
                LastName = "Mathews",
                Id = 12348,
                Test1Score = 87,
                Test2Score = 93
            };
            u1.AddStudent(s2);

            Student s3 = new Student
            {
                FirstName = "Adam",
                LastName = "Fredericks",
                Id = 12341,
                Test1Score = 82,
                Test2Score = 83
            };
            u1.AddStudent(s3);

            u1.SortStudent();  // uses Shell sort
            string out1 = "";
            foreach (Student st in u1.STList)
                out1 += st.ToString() + "\n";
            MessageBox.Show(out1);

            u1.SortStrategy = new QuickSortStrategy();
            u1.SortStudent();
            string out2 = "";
            foreach (Student st in u1.STList)
                out2 += st.ToString() + "\n";
            MessageBox.Show(out2);
        }
```

Template Pattern:

The purpose of the template algorithm is to define the different steps of an algorithm (template), however, leaving the actual implementation of the steps to subclasses. This allows for a flexible implementation of an algorithm that consists of well-defined sequence of steps. The UML for the template pattern is given below.



Create a new windows forms application project called TemplatePattern. Add a class called Student with the following code in it.

```
    class Student : IComparable
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Id { get; set; }
        public int Test1Score { get; set; }
        public int Test2Score { get; set; }
        public string Grade { get; set; }

        public int CompareTo(object obj)
        {
            int ret = 0;
            if (obj is Student)
            {
                ret = this.LastName.CompareTo(((Student)obj).LastName);
            }
            return ret;
        }

        public override string ToString()
        {
            return FirstName + " " + LastName + " : " + Id.ToString() +
                " : " + Test1Score.ToString() + " : " +
Test2Score.ToString();
        }
    }
```

Add a class called TemplateStudent which defines a method called ReadAndProcessStudents that defines the steps of an algorithm that are needed, as shown below.

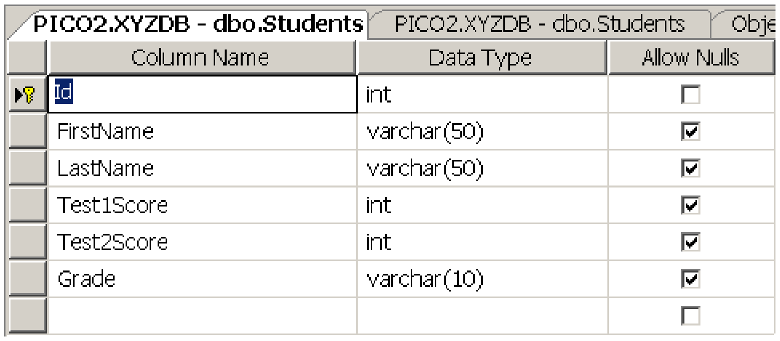
```
abstract class TemplateStudent  // base
{
    protected List<Student> _STList = new List<Student>();
    public List<Student> STList
    {
        get { return _STList; }
        set { _STList = value; }
    }

    public abstract void ReadStudents();
    public abstract void AssignGrades();
    public abstract void SortStudents();
    public abstract void StoreStudents();

    // algorithm steps
    public void ReadAndProcessStudents()
    {
        ReadStudents();  // from XML or comma delimited file, or DB
        AssignGrades();  // Formula may change
        SortStudents();  // Sort by Quick or Shell Sort, any field
        StoreStudents(); // Emit tab delimited file or XML, or DB
    }
}
```

Create a concrete class called StudentProcessingViaFile that inherits from the above TemplateStudent class. This class provides the code the abstract methods that form the steps of the template algorithm in the base class. The actual processing of the data is done on a file so the constructor expects an input file name and an output file name.

```
class StudentProcessingViaFile : TemplateStudent
{
    string inFileName = "";
    string outFileName = "";
    public StudentProcessingViaFile(string infName, string outfName)
    {
        inFileName = infName;
        outFileName = outfName;
    }

    public override void ReadStudents()
    {
        FileInfo fi = new FileInfo(inFileName);
        StreamReader sr = fi.OpenText();
        char[] seps = { ',' };
        string sline = sr.ReadLine();
        while (sline != null)
        {
            string[] parts = sline.Split(seps, 5);
            Student s1 = new Student();
            s1.FirstName = parts[0].Trim();
            s1.LastName = parts[1].Trim();
            s1.Id = int.Parse(parts[2]);
            s1.Test1Score = int.Parse(parts[3]);
            s1.Test2Score = int.Parse(parts[4]);
            _STList.Add(s1);
            sline = sr.ReadLine();
        }
        sr.Close();
    }
```

```
        public override void AssignGrades()
        {
            foreach (Student st in _STList)
            {
                string grade = "";
                double avg = 0.4 * st.Test2Score + 0.6 * st.Test2Score;
                if (avg > 90)
                    grade = "A";
                else if (avg > 85)
                    grade = "A-";
                else if (avg > 80)
                    grade = "B+";
                else
                    grade = "B";
                st.Grade = grade;
            }
        }

        public override void SortStudents()
        {
            _STList.Sort();  // could be Shell sort
        }

        public override void StoreStudents()
        {
            FileInfo fi = new FileInfo(outFileName);
            StreamWriter sw = new StreamWriter(fi.Open(FileMode.OpenOrCreate,
FileAccess.Write));
            foreach (Student st in _STList)
            {
                sw.WriteLine(st.FirstName + "\t" +
                    st.LastName + "\t" + st.Id.ToString() + "\t" +
                    st.Test1Score.ToString() + "\t" +
                    st.Test2Score.ToString() + "\t" + st.Grade);
            }
            sw.Close();
        }
    }
```

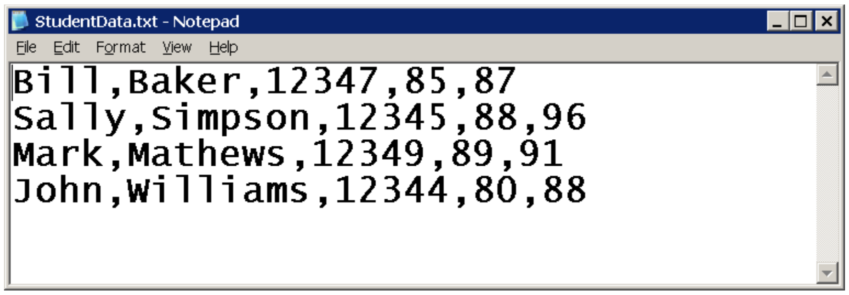
We will also provide a flexibility where the students data will come from the database. Create a database called XYZDB in SQL server (using SQL Server Management Studio). Then add a table called Students to it. The design of the Students table is shown below.



PICO2.XYZDB - dbo.Students | PICO2.XYZDB - dbo.Students | Obje

Column Name	Data Type	Allow Nulls
Id	int	☐
FirstName	varchar(50)	☑
LastName	varchar(50)	☑
Test1Score	int	☑
Test2Score	int	☑
Grade	varchar(10)	☑
		☐

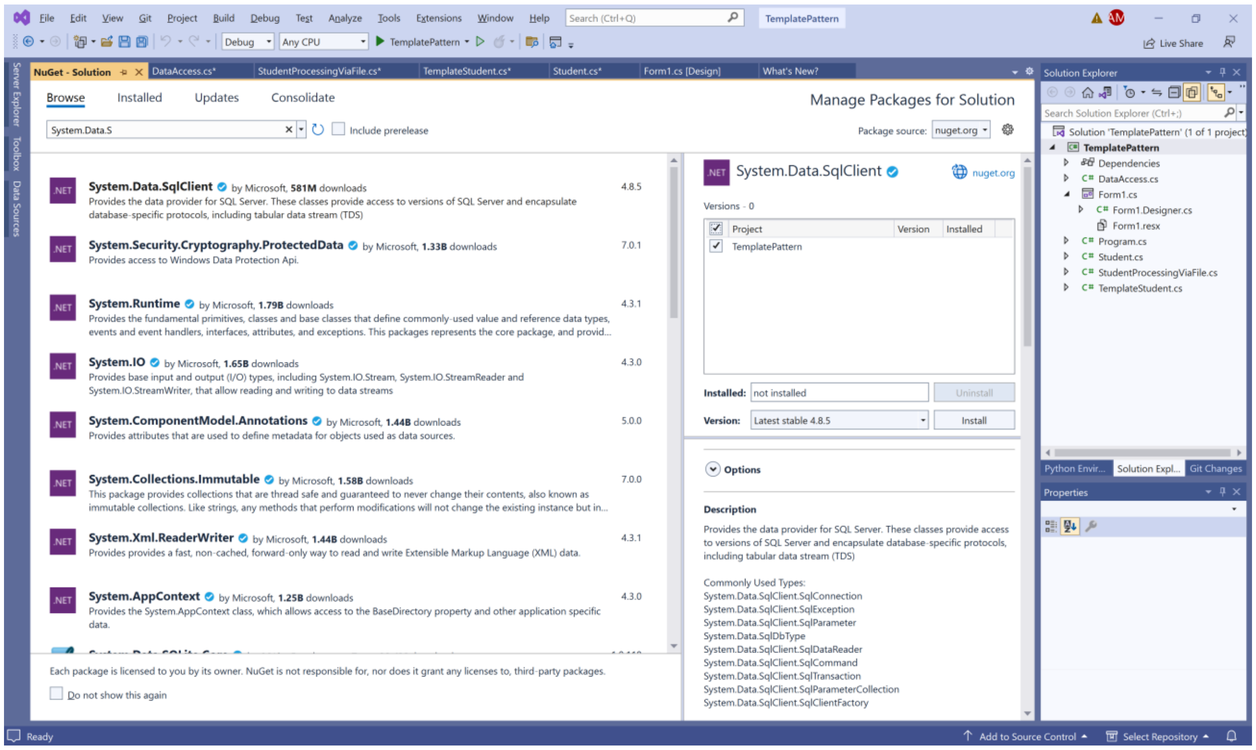
Prepare a text file called StudentData.txt with the following data in it. You can store it in your CPSC501 folder under the subfolder Data.



```
Bill,Baker,12347,85,87
Sally,Simpson,12345,88,96
Mark,Mathews,12349,89,91
John,Williams,12344,80,88
```

Store similar type of data in the Students table in the XYZDB database.

From the Tools -> Nuget Package Manager, choose Manage Nuget Packages for solution, then install System.Data.SqlClient package as shown below.



System.Data.S Include prerelease

Package source: nuget.org

System.Data.SqlClient by Microsoft, **581M** downloads 4.8.5
Provides the data provider for SQL Server. These classes provide access to versions of SQL Server and encapsulate database-specific protocols, including tabular data stream (TDS)

System.Security.Cryptography.ProtectedData by Microsoft, **1.33B** downloads 7.0.1
Provides access to Windows Data Protection Api.

System.Runtime by Microsoft, **1.79B** downloads 4.3.1
Provides the fundamental primitives, classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and exceptions. This packages represents the core package, and provid...

System.IO by Microsoft, **1.65B** downloads 4.3.0
Provides base input and output (I/O) types, including System.IO.Stream, System.IO.StreamReader and System.IO.StreamWriter, that allow reading and writing to data streams

System.ComponentModel.Annotations by Microsoft, **1.44B** downloads 5.0.0
Provides attributes that are used to define metadata for objects used as data sources.

System.Collections.Immutable by Microsoft, **1.58B** downloads 7.0.0
This package provides collections that are thread safe and guaranteed to never change their contents, also known as immutable collections. Like strings, any methods that perform modifications will not change the existing instance but in...

System.Xml.ReaderWriter by Microsoft, **1.44B** downloads 4.3.1
Provides provides a fast, non-cached, forward-only way to read and write Extensible Markup Language (XML) data.

System.AppContext by Microsoft, **1.25B** downloads 4.3.0
Provides the System.AppContext class, which allows access to the BaseDirectory property and other application specific data.

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

Do not show this again

System.Data.SqlClient nuget.org

Versions - 0

Project	Version	Installed
✓ TemplatePattern		

Installed: not installed

Version: Latest stable 4.8.5

Description

Provides the data provider for SQL Server. These classes provide access to versions of SQL Server and encapsulate database-specific protocols, including tabular data stream (TDS)

Commonly Used Types:
System.Data.SqlClient.SqlConnection
System.Data.SqlClient.SqlException
System.Data.SqlClient.SqlParameter
System.Data.SqlDbType
System.Data.SqlClient.SqlDataReader
System.Data.SqlClient.SqlCommand
System.Data.SqlClient.SqlTransaction
System.Data.SqlClient.SqlParameterCollection
System.Data.SqlClient.SqlClientFactory

Add a file to the project called DataAccess with the following database related code in it. Change the connection string below to match your database server. You will need to resolve the SqlConnection class and DataTable classes by right clicking on the name of class and choosing “quick actions” and then selecting the appropriate using statement to expose the namespace.

```
    class DataAccess
    {
        static string connStr = "server=pico2\\SQLExpress;integrated 
security=true;database=XYZDB";

        public static object GetSingleAnswer(string sql)
        {
            object res = null;
            SqlConnection conn = new SqlConnection(connStr);
            try
            {
                conn.Open();
                SqlCommand cmd = new SqlCommand(sql, conn);
                res = cmd.ExecuteScalar();
            }
            catch (Exception)
            {
                throw;
            }
            finally
            {
                conn.Close();
            }
            return res;
        }

        public static DataTable GetDataTable(string sql)
        {
            DataTable res = new DataTable();
            SqlConnection conn = new SqlConnection(connStr);
            try
            {
                conn.Open();
                SqlDataAdapter da = new SqlDataAdapter(sql, conn);
                da.Fill(res);
            }
            catch (Exception)
            {
                throw;
            }
            finally
            {
                conn.Close();
            }
            return res;
        }

        public static int InsertOrUpdateOrDelete(string sql)
        {
            int res = 0;
            SqlConnection conn = new SqlConnection(connStr);
            try
            {
                conn.Open();
                SqlCommand cmd = new SqlCommand(sql, conn);
                res = cmd.ExecuteNonQuery();
            }
            catch (Exception)
            {
                throw;
            }
            finally
            {
                conn.Close();
            }
            return res;
        }
    }
```

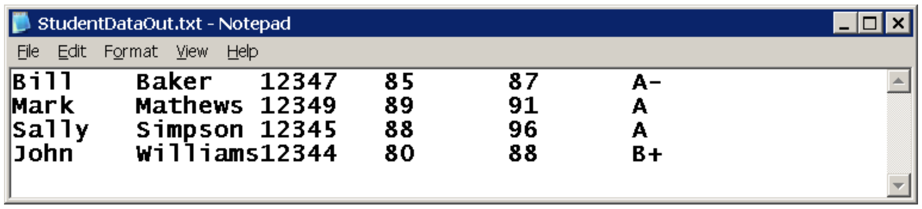
To provide an implementation of the algorithm that uses a different grading formula and also operates on data from a database, add a class called StudentProcessingViaDB with the following code in it.

```
    class StudentProcessingViaDB : TemplateStudent
    {

        public override void ReadStudents()
        {
            string sql = "select * from Students";
            DataTable dt = DataAccess.GetDataTable(sql);
            for (int i = 0; i < dt.Rows.Count; i++)
            {
                Student s1 = new Student();
                s1.FirstName = dt.Rows[i]["FirstName"].ToString();
                s1.LastName = dt.Rows[i]["LastName"].ToString();
                s1.Id = int.Parse(dt.Rows[i]["Id"].ToString());
                s1.Test1Score =
int.Parse(dt.Rows[i]["Test1Score"].ToString());
                s1.Test2Score =
int.Parse(dt.Rows[i]["Test2Score"].ToString());
                _STList.Add(s1);
            }
        }

        public override void AssignGrades()
        {
            foreach (Student st in _STList)
            {  // different formula
                string grade = "";
                double avg = 0.5 * st.Test2Score + 0.5 * st.Test2Score;
                if (avg > 90)
                    grade = "A";
                else if (avg > 85)
                    grade = "A-";
                else if (avg > 80)
                    grade = "B+";
                else
                    grade = "B";
                st.Grade = grade;
            }
        }

        public override void SortStudents()
        {
               // doesn't matter because data is being written to DB
        }

        public override void StoreStudents()
        {
            foreach (Student st in _STList)
            {
                string sql = "Update Students set " +
                    "FirstName='" +  st.FirstName + "'," +
                    "Lastname='" + st.LastName + "'," +
                    "Test1Score=" + st.Test1Score + "," +
                    "Test2Score=" + st.Test2Score + "," +
                    "Grade='" + st.Grade + "' where Id=" + st.Id.ToString();
                DataAccess.InsertOrUpdateOrDelete(sql);
            }
        }
    }
```

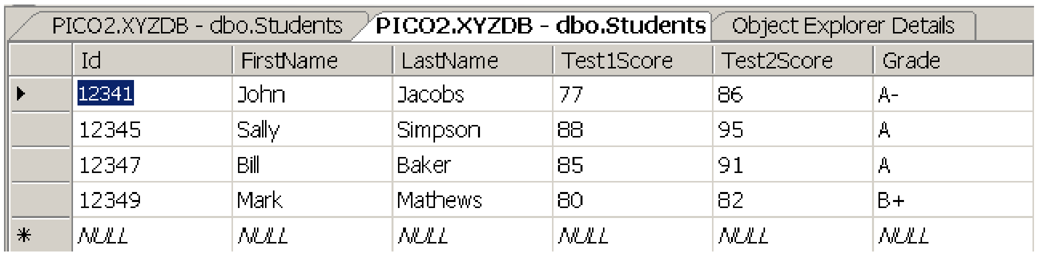
To test the template pattern, add a button called btnTemplate with the following code in its handler.

```
        private void btnTemplate_Click(object sender, EventArgs e)
        {
            TemplateStudent tst = new StudentProcessingViaFile(
        @"c:\CPSC501\Data\StudentData.txt",
@"c:\CPSC501\Data\StudentDataOut.txt");
            tst.ReadAndProcessStudents();

            // process students via DB
            TemplateStudent tst2 = new StudentProcessingViaDB();
            tst2.ReadAndProcessStudents();

            MessageBox.Show("done processing..");
        }
```

If you run the above program, it will process the student data from the text file and also from the database. The output in the StudentDataOut.txt and the Students table in the database is shown below.



```
Bill    Baker   12347   85      87      A-
Mark    Mathews 12349   89      91      A
Sally   Simpson 12345   88      96      A
John    Williams12344   80      88      B+
```



	Id	FirstName	LastName	Test1Score	Test2Score	Grade
▶	12341	John	Jacobs	77	86	A-
	12345	Sally	Simpson	88	95	A
	12347	Bill	Baker	85	91	A
	12349	Mark	Mathews	80	82	B+
*	NULL	NULL	NULL	NULL	NULL	NULL

What if the grading formula needed to be changed quite often. To accommodate minimal changes to the implementation steps of the template algorithm, we can implement the strategy pattern for a step of the algorithm and thus provide a pluggable implementation of a step. To demonstrate this, add an interface called IGradeStrategy with the following code in it.

```
    interface IGradeStrategy
    {
        string ComputeGrade(Student st);
    }
```

Provide two concrete strategy classes for computing the grade called ComputeGrade4060 and ComputeGrade5050 with the following codes in them.

```
    class ComputeGrade4060 : IGradeStrategy
    {

        public string ComputeGrade(Student st)
        {
            string grade = "";
            double avg = 0.4 * st.Test1Score + 0.6 * st.Test2Score;
            if (avg > 90)
                grade = "A";
            else if (avg > 85)
                grade = "A-";
            else if (avg > 80)
                grade = "B+";
            else
                grade = "B";
            return grade; ;
        }
    }


    class ComputeGrade5050 : IGradeStrategy // concrete strategy
    {

        public string ComputeGrade(Student st)
        {
            string grade = "";
            double avg = 0.5 * st.Test1Score + 0.5 * st.Test2Score;
            if (avg > 90)
                grade = "A";
            else if (avg > 85)
                grade = "A-";
            else if (avg > 80)
                grade = "B+";
            else
                grade = "B";
            return grade;
        }
    }
```

As you can see from the above code, one strategy implements a weight of 40% and 60% for the two test scores in computing the average, and the other gives them equal weight of 50% each. The changes made with respect to the StudentProcessingViaFile class are shown in bold.

Add a class called StudentProcessingViaFile2 with the following code in it.

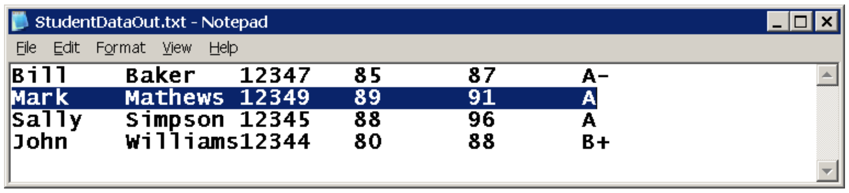
```
class StudentProcessingViaFile2 : TemplateStudent
{  // combining template and strategy patterns to make
   // part of the algorithm step pluggable.
    string inFileName = "";
    string outFileName = "";
    IGradeStrategy igradeStrategy;  // flexible grade strategy
    public StudentProcessingViaFile2(string infName, string outfName,
        IGradeStrategy igr)
    {
        inFileName = infName;
        outFileName = outfName;
        igradeStrategy = igr;
    }

    public override void ReadStudents()
    {
        FileInfo fi = new FileInfo(inFileName);
        StreamReader sr = fi.OpenText();
        char[] seps = { ',' };
        string sline = sr.ReadLine();
        while (sline != null)
        {
            string[] parts = sline.Split(seps, 5);
            Student s1 = new Student();
            s1.FirstName = parts[0].Trim();
```

```
                s1.LastName = parts[1].Trim();
                s1.Id = int.Parse(parts[2]);
                s1.Test1Score = int.Parse(parts[3]);
                s1.Test2Score = int.Parse(parts[4]);
                _STList.Add(s1);
                sline = sr.ReadLine();
            }
            sr.Close();
        }

        public override void AssignGrades()
        {
            foreach (Student st in _STList)
            {
                st.Grade = igradeStrategy.ComputeGrade(st);
            }
        }

        public override void SortStudents()
        {
            _STList.Sort();  // could be Shell sort
        }

        public override void StoreStudents()
        {
            FileInfo fi = new FileInfo(outFileName);
            StreamWriter sw = new StreamWriter(fi.Open(FileMode.OpenOrCreate,
FileAccess.Write));
            foreach (Student st in _STList)
            {
                sw.WriteLine(st.FirstName + "\t" +
                    st.LastName + "\t" + st.Id.ToString() + "\t" +
                    st.Test1Score.ToString() + "\t" +
                    st.Test2Score.ToString() + "\t" + st.Grade);
            }
            sw.Close();
        }
    }
```

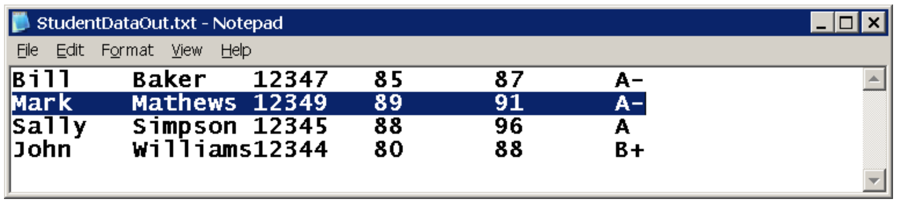
To test the above combination of template and strategy patterns, add a button to the form called btnTemplateStrategy with the following code in it.

```
  private void btnTemplateAndStrategy_Click(object sender, EventArgs e)
        {
            IGradeStrategy igst = new ComputeGrade4060();
            //IGradeStrategy igst = new ComputeGrade5050();
            TemplateStudent tst = new StudentProcessingViaFile2(
                 @"c:\CPSC501\Data\StudentData.txt",
@"c:\CPSC501\Data\StudentDataOut.txt",igst);
            tst.ReadAndProcessStudents();
            MessageBox.Show("done..");
        }
```

If you run the program and click on the above button, you will see Mark Mathews is given a grade of A if the ComputeGrade4060 strategy is provided, otherwise a grade of A- if the ComputeGrade5050 strategy is used.



```
Bill     Baker    12347    85       87       A-
Mark     Mathews  12349    89       91       A
Sally    Simpson  12345    88       96       A
John     Williams12344    80       88       B+
```



```
Bill     Baker    12347    85       87       A-
Mark     Mathews  12349    89       91       A-
Sally    Simpson  12345    88       96       A
John     Williams12344    80       88       B+
```