|  |
| --- |
| Ofiaich College |
| Visual Basic Assignment 02 |
| Student Records |
|  |
| **Paul Millar** |
| **8/1/2012** |

|  |
| --- |
|  |

Contents

[User Interface Snapshot 3](#_Toc313895492)

[Introduction 4](#_Toc313895493)

[Helper Objects 9](#_Toc313895494)

[*‘Grades.vb’* 9](#_Toc313895495)

[*‘SearchOptions.vb’* 10](#_Toc313895496)

[*‘’StudentDataNames.vb’* 11](#_Toc313895497)

[User Controls 13](#_Toc313895498)

[Templates 13](#_Toc313895499)

[The Button Template – ‘*tplButton.vb’* 13](#_Toc313895500)

[The Combo Box Template – ‘*tplComboBox.vb*’ 15](#_Toc313895501)

[The Divider Template – ‘*tplDivider.vb’* 16](#_Toc313895502)

[How The Interface Is Built 17](#_Toc313895503)

[The Main Form - *‘frmStudent.vb’* 20](#_Toc313895504)

[The Header – ‘*ucStudentHeader.vb’* 21](#_Toc313895505)

[The Toolbar – ‘*ucStudentToolbar.vb’* 22](#_Toc313895506)

[The Data Grid – ‘*ucStudentDataGrid.vb’* 27](#_Toc313895507)

[The Information Panel – ‘*ucStudentTotals.vb’* 30](#_Toc313895508)

[Automating Information Updates Using A Custom Event. 31](#_Toc313895509)

[Relevant Information 33](#_Toc313895510)

[LINQ Queries 35](#_Toc313895511)

[Outputting Information 37](#_Toc313895512)

[Data Objects 39](#_Toc313895513)

[The Student Data Table - *‘dtStudent.vb’* 39](#_Toc313895514)

[Validation 41](#_Toc313895515)

[Calculating the Overall Mark & Assigning The Grade 42](#_Toc313895516)

[Distinct Addresses - ‘*dtDistinctAddresses.vb’* 44](#_Toc313895517)

[Distinct Classes - *‘dtDistinctClasses.vb’* 45](#_Toc313895518)

[The DataSet – ‘*dsStudent.vb’* 46](#_Toc313895519)

[The Data Adapter – *‘adprStudent.vb’* 47](#_Toc313895520)

[Updating 49](#_Toc313895521)

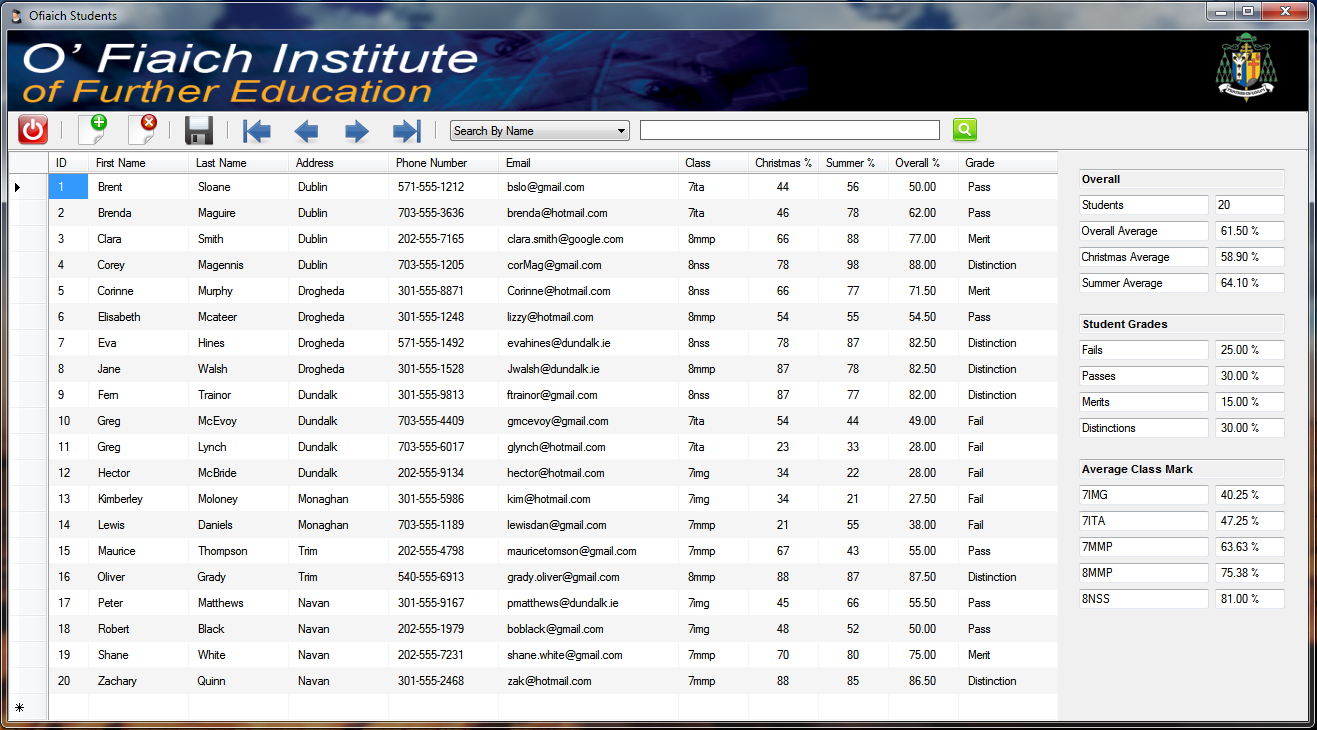
[Selecting Student Records 51](#_Toc313895522)

[Binding 55](#_Toc313895523)

[Problems 57](#_Toc313895524)

# 

# User Interface Snapshot



# 

# Introduction

This project was developed with Visual Studio 2010 Ultimate, using .NET Framework 3.5.

***Features***

* Adding Student Records.
* Deleting Student Records.
* Saving Student Records.
* Searching Student Records.
  + By Name
  + By Address
  + By Class
  + By Grade
* Navigating Student Records.
* Information (*Totals & Averages*)
  + Total Number Of Students.
  + Overall Christmas Average.
  + Overall Summer Average.
  + Overall Mark Average.
  + Overall Grade Rates.
  + Average Class Marks.

Additional columns were also added to the Data Grid that display a student’s overall mark plus the grade earned.

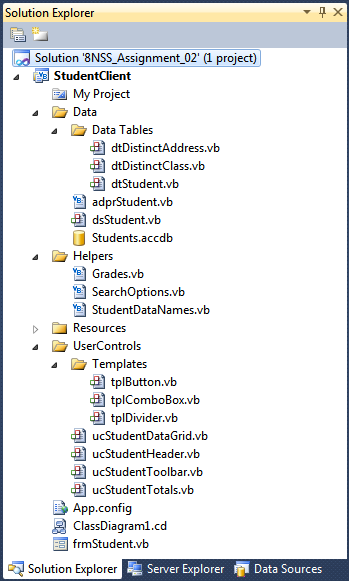
For this solution, all objects were hard coded and are grouped into three categories...

* Helper Objects - Page 9
* User Interface Objects - Page 13
* Data Objects - Page 39

As there is quite a bit to this, the following are pointers to the more relevant parts as required by the projects specifications.

* SQL Statements - Pages 42, 44, 51 - 52
* Calculating Averages & LINQ - Pages 35 - 36
* Searching - Pages 10, 25 – 26, 51 – 54
* Updating - Pages 49 - 50

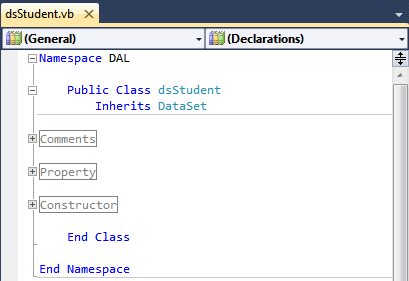
Here’s how the overall solution shapes up.



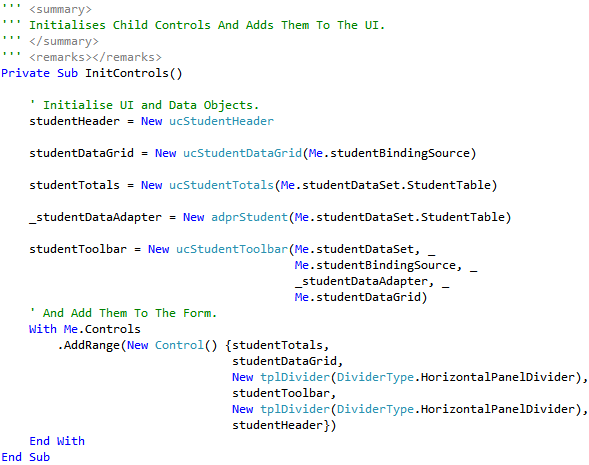
#### Comments

Comments are found throughout the solution. Every class has a comment to briefly explain its functionality, whilst every class member also has an xml comment.

Class comments can be found in the ‘*Comments’* region. This is the first region to be seen in a class as shown here.

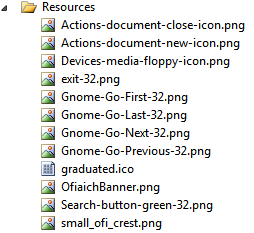


You will find an xml comment located above each and every class member and in some cases, comments can be also be found above statements. (Example from ‘*frmStudent.vb’*  - the main form)

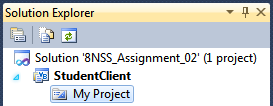


#### Resource Files (Images and Icons)

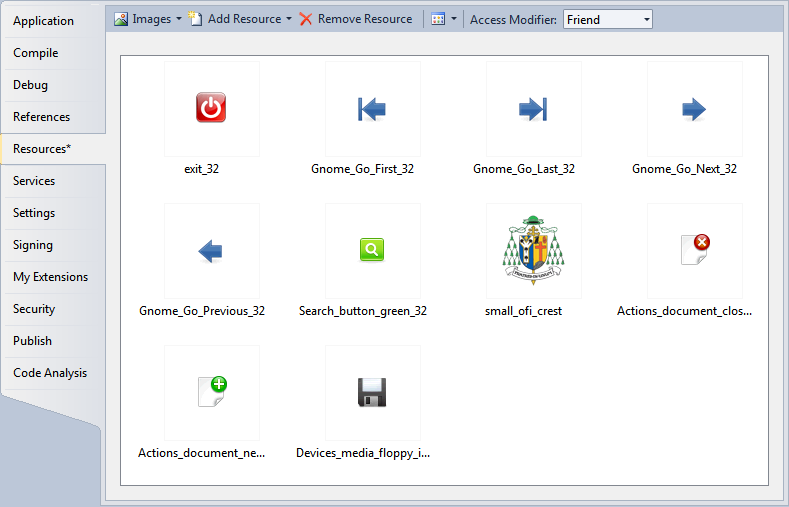
To use images and icons, we need to add them to the project as resources. This solution uses the following files.



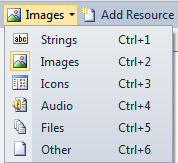
In order to add them to the solution, first double click on the ‘My Project’ node in ‘Solution Explorer’



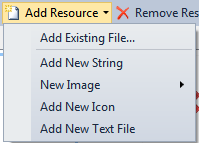
From the tabs on the left, select the ‘Resources’ tab and you should be looking at the following screen.



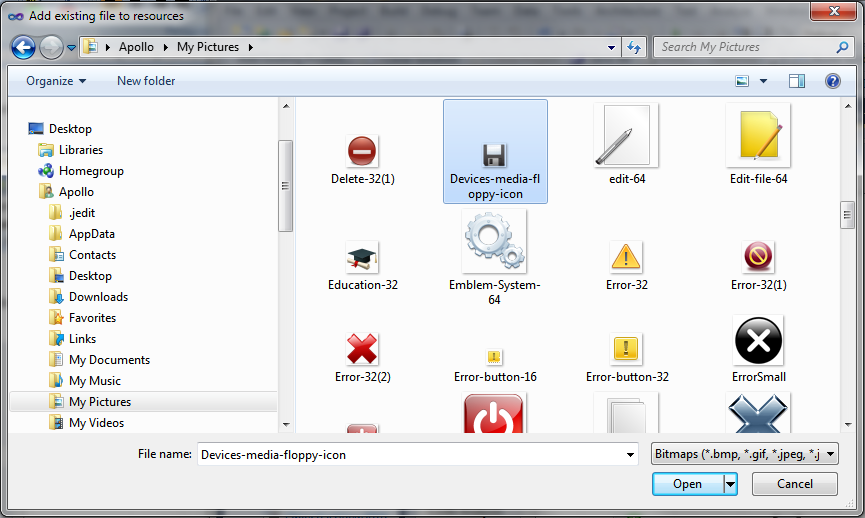
To choose what type of resource you want to add to the solution, click on the first combo box and choose from the types offered.



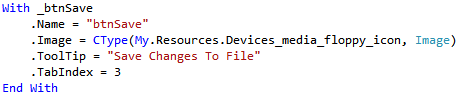
To add the resource, choose ‘Add Existing Fie’ from the ‘Add Resource’ combo box.



From the following ‘Explorer’ window, simply select the file you want and click ‘open’.

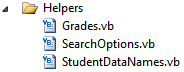


The following code shows you how to apply an image to a button control.

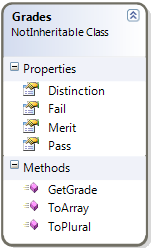


# Helper Objects

These are simply a collection of static *classes* ad *enums* used throughout the solution.



## *‘Grades.vb’*

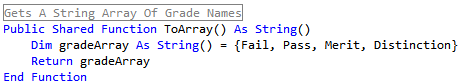


*This is a static class that determines the appropriate Grade for a given exam mark.*

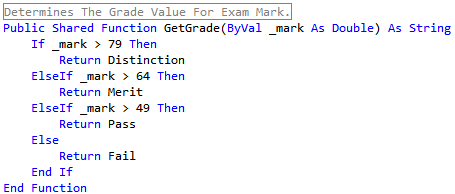
*It also provides a list of possible grades for use when binding to controls that provide us with options for filtering student records.*

This class contains four *String* properties that return the different grades; ‘Fail’, ‘Pass’, ‘Merit’ and ‘Distinction’.

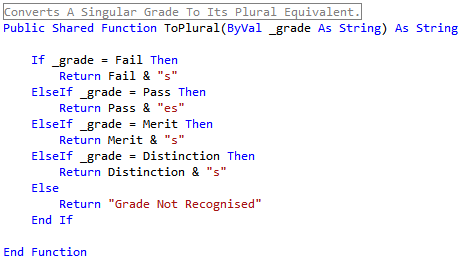
The function ‘*ToArray*’, returns an Array(Of String) that contains all grades.



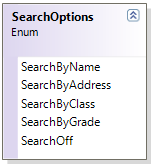
The function ‘*GetGrade’*, determines the appropriate grade for a given mark and returns a String value containing the grade.



The function ‘*ToPlural’*, returns the pluralised equivalent of a grade. The signature takes a string argument that specifies that singular version of the grade, e.g. ‘Fail’ (will return ‘Fails’).

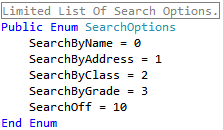


## *‘SearchOptions.vb’*



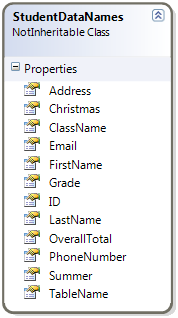
*This enum provides a list of possible search options used by the solution.*

The values are...



## *‘’StudentDataNames.vb’*

Being aware from the beginning that we would have to code the actual columns names from the ‘*StudentRecord’* database table, into a multitude of *SQL SELECT* statements in *command* objects, plus naming data grid columns and table columns, this class was created to provide consistency throughout the solution. Essentially we wanted to reduce the possibility of errors.

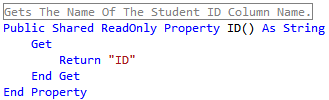


*This is a static class that simply returns the actual column names used in the Access database.*

*It’s purpose is to reduce typo errors in our code and to provide consistency when binding.*

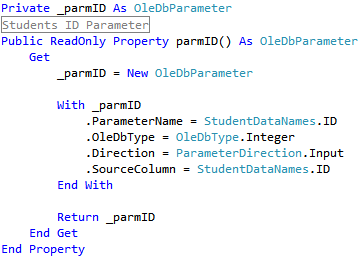
*The class contains 11 read only String properties that return the name of each column and one that returns the* *name of the database table.*

This is an example demonstrating the *ID* property.

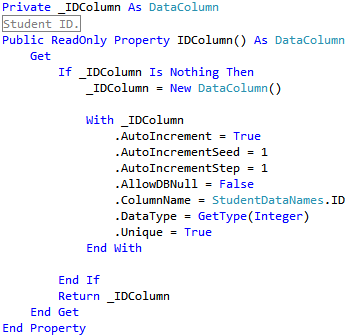


The following three examples show how we implement this class when binding.

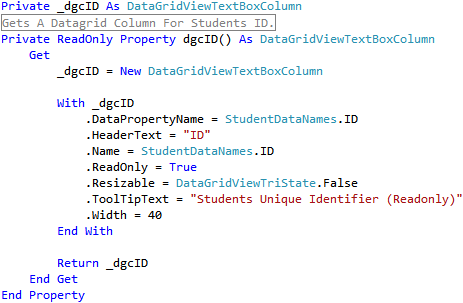
When creating a parameter used by the student data adapter...



Creating a Data Column for the student Data Table...



Finally, creating a Data Grid Column for the student Data Grid...

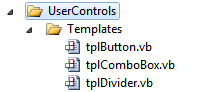


# User Controls

## Templates

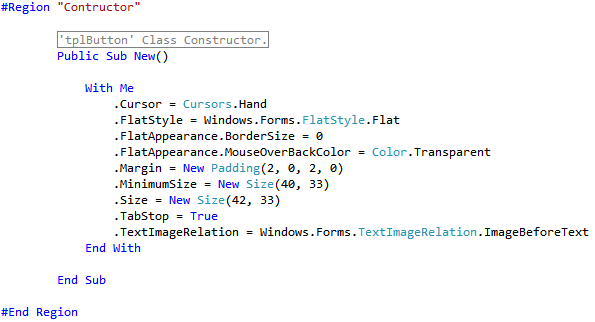
For a consistent look throughout the application, all buttons are derived from the ‘*tplButton’* class and Combo Boxes derive from the ‘*tplComboBox*’ class. Another template ‘tplDividers’, creates dividers that separate the different controls on the main form and button control sections on the toolbar. These classes can be thought of as CSS files.

In Solution Explorer, Templates can be found under the following folders...



### The Button Template – ‘*tplButton.vb’*

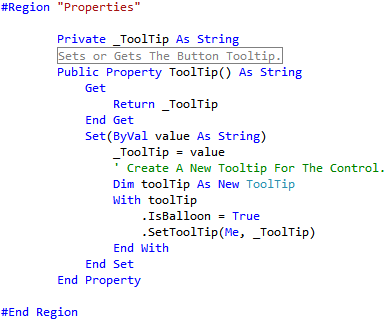
This template inherits from the ***Button*** class. It’s constructor sets out the basic appearance of the button and it’s behaviour when the user hovers the mouse over it.



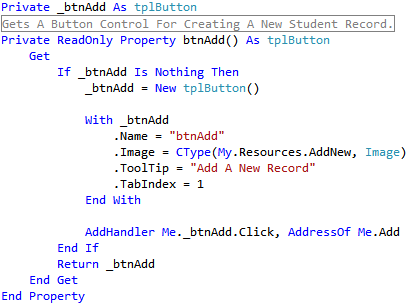
*Note:* All button controls are created in the ‘*ucStudentToolbar*’ class.

#### The Button ToolTip

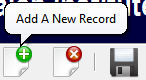
With Windows Forms, you have to create a *Tooltip* object, assign the text and then assign it to the button control that is going to display it. This template has a string property that creates a tooltip for you simply by *Set*ting its value.



When the ‘Add’ button is instantiated ...

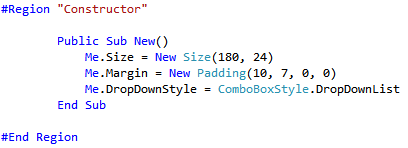


The tooltip looks like the following...

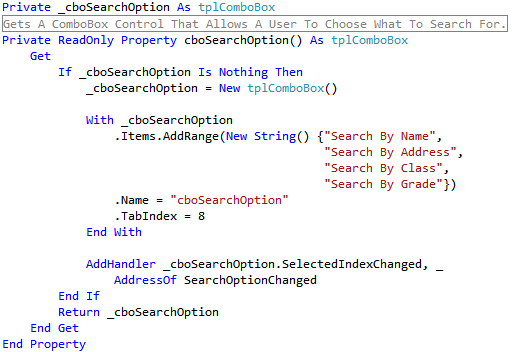


### The Combo Box Template – ‘*tplComboBox.vb*’

This template inherits from the ***ComboBox*** class. Its constructor sets out the basic appearance we want for our combo boxes.



There are two combo boxes, the following code creates a new ‘C*omboBox’* that is used for displaying the different search options available. The code for both combo boxes is available in the ‘*ucStudentToolbar.vb’* class.

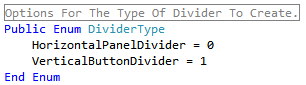


After adjusting the size of the second combo box, the result is...

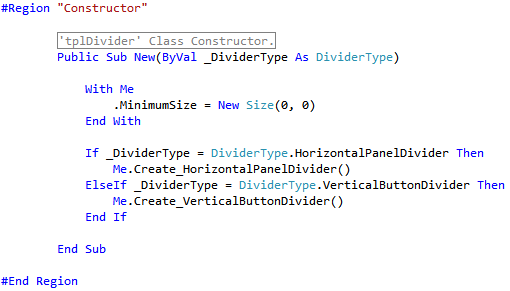


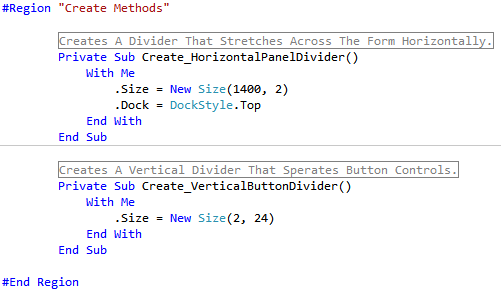
### The Divider Template – ‘*tplDivider.vb’*

This template inherits from the ***GroupBox*** class. An enum is declared to provide different options for the type of divider the developer wants to create.



This option is passed as an argument to the constructor which then calls the appropriate create method.





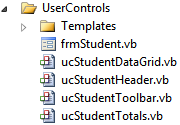
Here’s how both dividers looks...

Horizontal >>  Vertical >> 

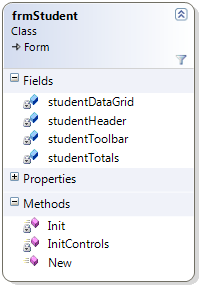
## How The Interface Is Built

Before we begin, it should be noted that all parent controls are responsible for creating their own child controls and for the handling of their events.

There are five ‘*parent’* user controls.



##### Class Representations



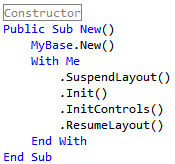
All of these components have at least 3 methods used for creating and initialising each component. They are ***New, Init, InitControls***. Because the principles and overall functionality of these methods are exactly the same for each component, we will discuss these first and then move on later to explain each component separately, only explaining any additional features.

#### Constructors

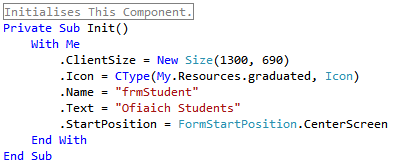
When constructing a user control, there are three methods that are involved in the process. For this report we will refer to them as *constructor* *methods*.

All examples are taken from the ‘*frmStudent.vb’* class

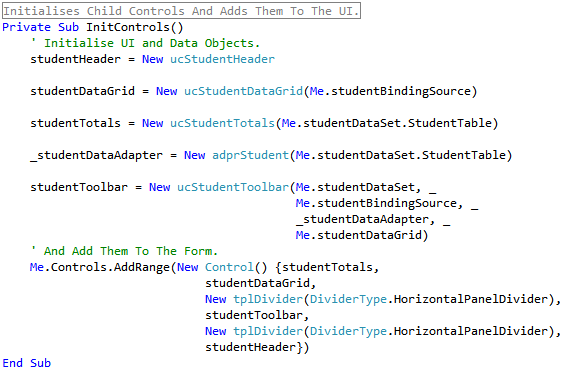
**New**: The Class Constructor deals with any arguments passed , Suspends and Resumes layout for performance reasons, and calls the ***Init*** and ***InitControls*** methods.



**Init:** Initialises the parent component (In this case, the form itself).

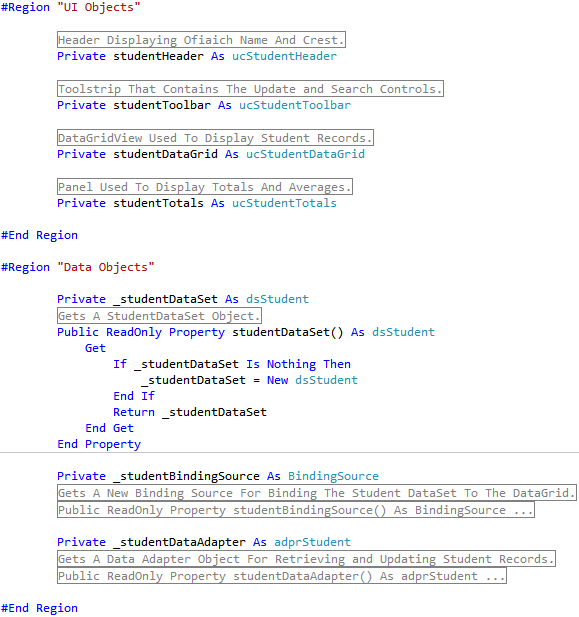


**InitControls:** Initialises and adds any child controls to the parent component .



#### Child controls

Child controls are provided by way of a class variable or property that returns an object of the control type we need.

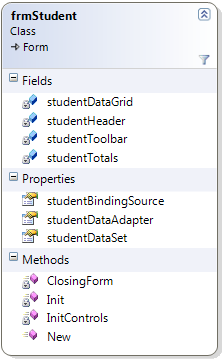


## The Main Form - *‘frmStudent.vb’*

This component inherits from ***Form*** class. All major objects (UI and Data) are created here...

The ‘Main Form’, creates and adds four ‘*child’* controls;the Header, Toolbar, Data Grid and the Information Panel, to the UI.

It also creates the main data objects required throughout the solution by each of the child UI components mentioned above. These data objects are passed, *By Reference*, to the UI components.



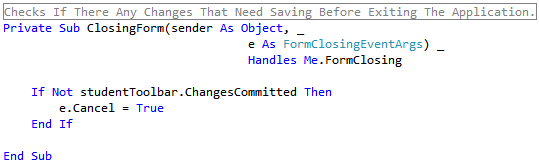
*This is the applications ‘Main Form’.*

*It instantiates and adds four ‘child’ controls to the UI.*

*It also creates three Data objects and passes them ‘By Reference’ to the UI components.*

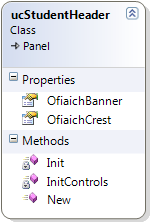
As previously explained, we use three methods to construct initialise the main form component and add its child controls.

This class has one other member that handles the ‘*FormClosing’* event. This checks for any changes in our data set and offers the user the chance to save them before closing the application down.



The ‘*ChangesCommitted’* function is located in the ‘*ucStudentToolbar.vb’* class, and it is this method that actually asks the user on how they want to proceed. The data type returned is *Boolean*.

## The Header – ‘*ucStudentHeader.vb’*



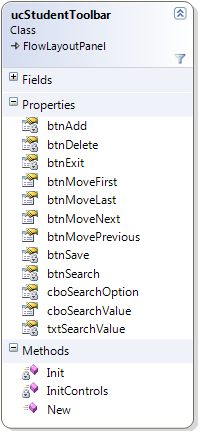
*This component inherits from the* ***Panel*** *class.*

*It simply contains two* ***PictureBox*** *controls that are added to the panel.*

Here’s the result...



## The Toolbar – ‘*ucStudentToolbar.vb’*



*This component inherits from the ‘****FlowLayoutPanel’*** *class.*

*It deals with the Updating off, Navigating through and Searching for student records.*

*A* ***FlowLayoutPanel*** *was chosen for its ease of use in that you can simply add multiple controls to it and it will automatically position them for you.*

This is how the toolbar looks when constructed...



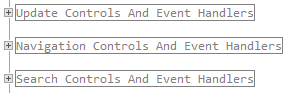
As with the previous controls, the ‘*InitControls’* method initialises and adds our child controls. There are nine button controls, two combo boxes and one text box control.

#### Region Hierarchy

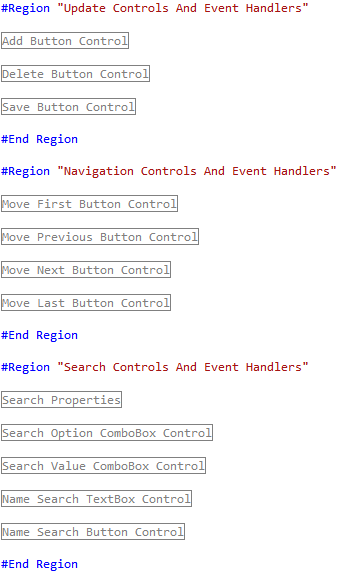
This is probably the most complex class in the solution as it deals with updating, navigating and searching student records, which resulted in a high volume of code. For that reason a hierarchy of regions was created to group related functions.

Three of the regions within this class deal with the main aspects of this component; updating, navigating and searching. Each region creates its own controls and handles their respective events.

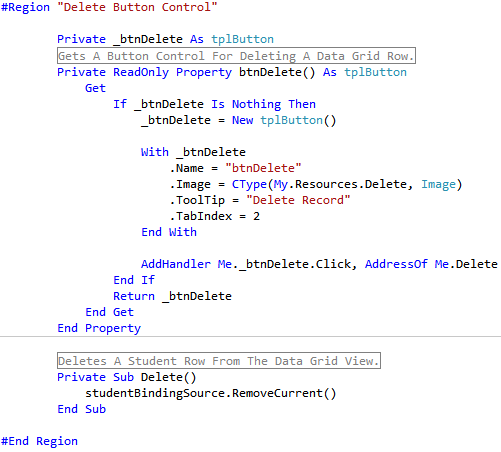
Here are the regions contracted...



To navigate to a particular control we expand these regions...



We then expand a control region....



The private variable ‘\_*btnDelete’*, could have been declared with the ‘*WithEvents’* Keyword, but since we are only concerned with one event, the ‘*Click’* event, a custom method called ‘*Delete’* was created to handle this.

We point the buttons ‘*Click’* event to this handler with the following code...



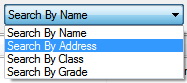
We also set the ‘*TabIndex’* property of each control so that focus will shift from the left most control to the right most when the user presses the *Tab* Key.

The same principles described here are applied to all button and combo box controls.

#### Search Controls

Not all controls are created by the constructor methods, some are created when the search option changes.

There are four different search options available as shown...



When searching ‘By Name’, a *TextBox* and a *Button* control is displayed, allowing the user to key in a ***search value***.



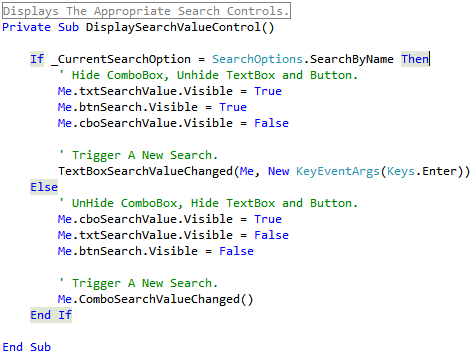
When searching by Address, Class or Grade, another *ComboBox* control is displayed that allows the user to select a *search* *value*.



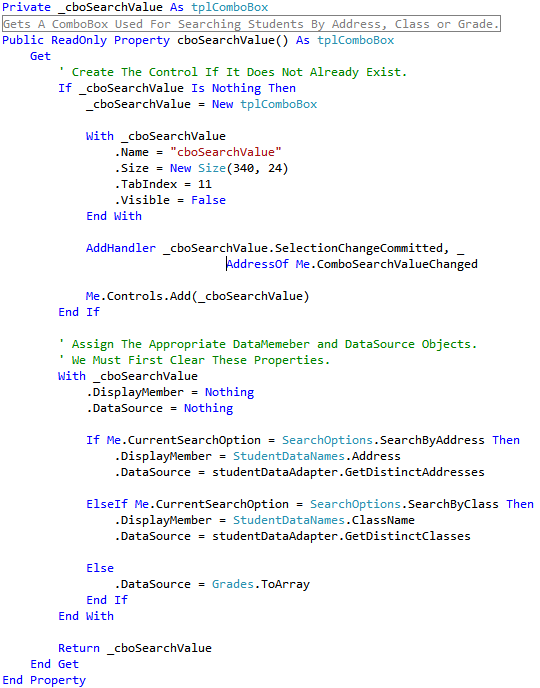




The ‘*SearchOptionChanged’* method handles the ‘*SelectedIndexChanged’* event for the search option combobox control. When executed, it in turn calls the ‘*DisplaySearchValueControl’* method which contains the following code that displays the appropriate controls...

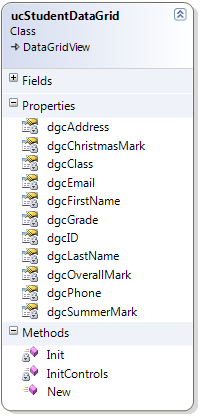


It should be pointed out that the same combo box is used for displaying the addresses, classes and grades that are used for filtering student records, we simply changed its ‘*DataSource’* and ‘*DataMember’* properties.



We use the student data adapter to get a list of distinct addresses and classes, however, the grades are sourced from a static class called *Grades* (located in the *Helpers* folder) that has four *String* properties representing the different grades. This class has a function ‘*ToArray’* that returns an Array(Of String) containing the different grades.

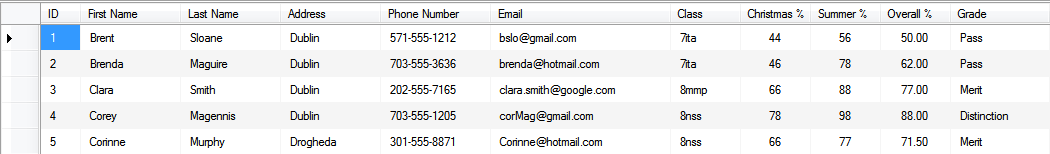
## The Data Grid – ‘*ucStudentDataGrid.vb’*



*This component inherits from the ‘****DataGridView****’* *class and is used to displays our student records.*

The class contains eleven properties of type ‘*DataGridViewTextBoxColumn’* , which are added to the data grid through its constructor methods. Certain column cells are then formatted for alignment purposes. The following pages show examples of how columns are created, added and formatted.

When created, the control looks like the following.



As you can see, we have additional columns for the *Overall Mark* and *Grade* of each student.

The class constructor has a parameter of type – ‘*BindingSource’* that is passed ‘*By Reference’.* This is assigned as the data grids ‘*DataSource’*.

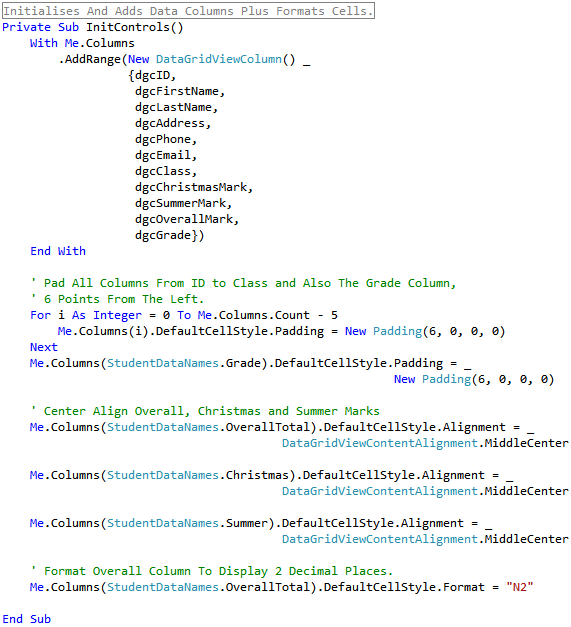
All ‘*DataGridViewColumns’* are of type ‘*DataGridViewTextBoxColumn’*. The following examples show how the ‘*ID’* and ‘*FirstName’* columns are created.



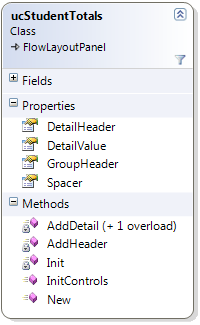
You can see from above that we also assigned a tooltip to each column...



Here, we see how columns are added and cells formatted...



## The Information Panel – ‘*ucStudentTotals.vb’*



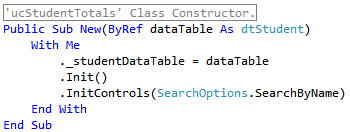
*This component inherits from the ‘****Panel****’* *class and is used to displays our Totals and Average calculations based on the current student records being viewed.*

All calculations are processed dynamically, using the records that are currently in the student data table (even when filtered). For that reason, we pass, *By Reference*, a student data table object, instantiated in the main student form, to this class’s constructor.

In the main student form we pass in the data table...



The class constructor assigns the argument to a local variable...



This is the variable declaration with the ‘*WithEvents’* keyword, for which the reason is explained next.

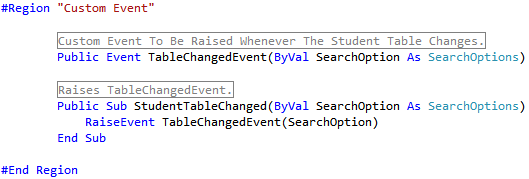


### Automating Information Updates Using A Custom Event.

For the information to be relevant to the records being viewed in the data grid, we need to recalculate averages every time we either save changes to, or fill the student data table.

Like this class, the data adapter class ‘*adprStudent.vb*’, is also passed, *By Reference*, a student data table object, and because this is where all updates and fills take place, it seems like the most logical place to initiate a re-calculation. But how does the data adapter communicate with the information panel, an object it knows nothing about.

The solution comes with the fact that both these objects share a common object, the student data table, and it is in that class that we declare a custom event.



As you can see, the event takes an argument of type, *SearchOption*. This will be explained fully later but basically, the calculations and information displayed are relevant to how the student records are filtered.

As you can see, the ‘*StudentTableChanged’* method raises the event.

In the data adapter, we have one method that updates the student data table, and two that populate the data table.



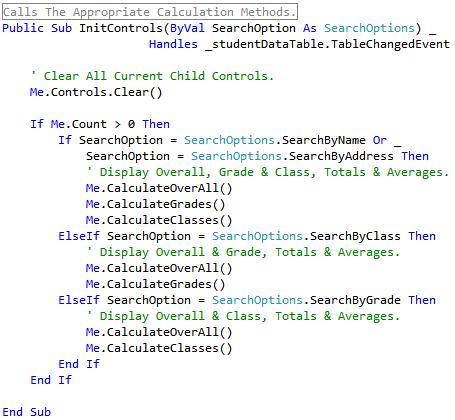




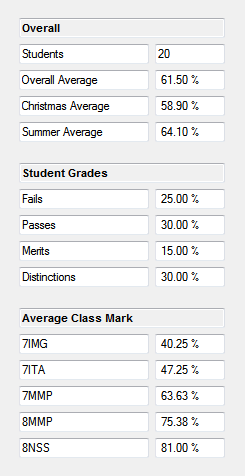
All three methods raise the ‘*TableChangedEvent’* by calling the ‘*StudentTableChanged’* method.



In the information panel class ‘*ucStudentTotals.vb’*, we have a method that handles the event any time it is raised, performs the appropriate calculation and creates the relevant child controls to display the calculations (for this reason we use the *InitControls* method).



### Relevant Information



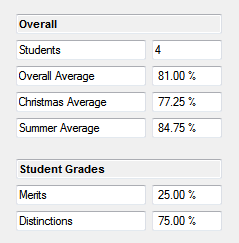
The ***Average Class Mark*** group calculates the overall mark for each class. Again, these calculations are based on the students currently being viewed.

The ***Student Grades*** group calculates, as percentage of 100, how many students failed, passed or earned merits or distinctions. Again, these calculations are based on the students currently being viewed.

The ***Overall*** group counts the number of students currently being viewed, and calculates the average exam marks for these students.

The information displayed above is only displayed when the end user searches for students by name or address.

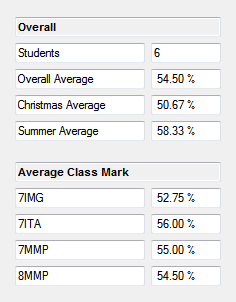
When we search for students by class, only the following information is displayed. (Search on ‘8NSS’)



This information tells us that the overall mark for the four students in 8NSS, is 81% and that 25% of the class earned merits while the other 75% earned distinctions.

*Think about it, we are filtering by a* ***single*** *class (8NSS) and thus, everything we need to know is located in the* ***Overall*** *group.*

When we search by grade, the same principle is applied, and only the following information is displayed when we search for students who ‘Passed’ their exams.

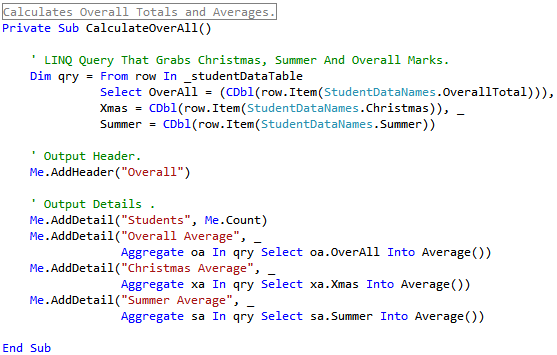


The ***Average Class Mark*** group tells us the average ‘***Pass’*** mark for each class.

The ***Overall*** group tells us that 6 students passed and that their average mark was 54.5 %.

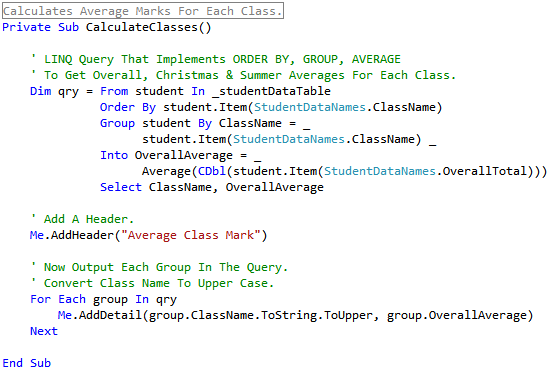
### LINQ Queries

For calculating the *Overall* group averages we use the following query.



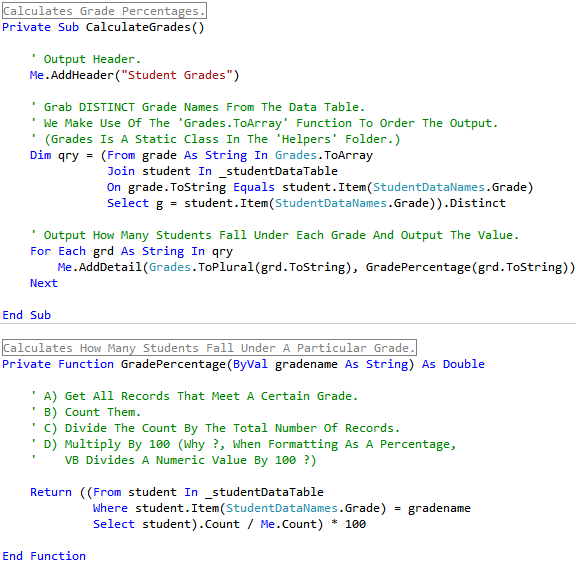
Essentially what we are doing here is using a LINQ query to grab the columns with student marks (Christmas, Summer and Overall), from the records currently in the students data table. We then use three more queries to calculate the average marks for each column.

Calculating the average mark of each class requires us to *Group* all the students by class and then calculate the average for each group. We then iterate through the query and output the results.



Calculating grade averages is a little more trickier because we need to first, get distinct grades in the student data table, and then call a function that runs another query to get the averages for each grade.

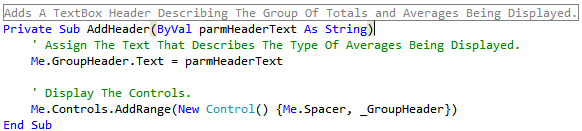
Because grades can appear in any order in the student table, we make use of the ‘*Grades.ToArray’* function and a ‘*Join’* statement to order the output, i.e. Fails, Passes, Merits, Distinctions.



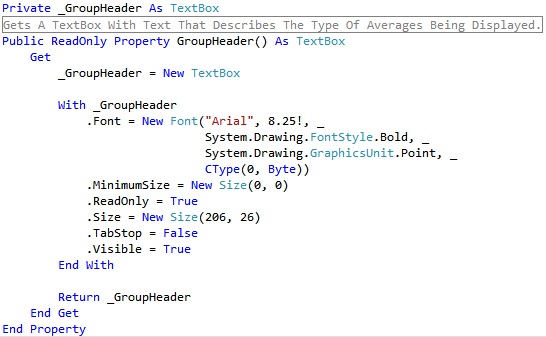
### Outputting Information

In all calculations you will have noticed two other methods, namely ‘*AddHeader*’ and ‘*AddDetail*’.

‘***AddHeader’*** is a method used to display the group headers. We simply pass in the header title.



‘*GroupHeader’* is a propertyof type *TextBox*.



When we call *AddHeader*



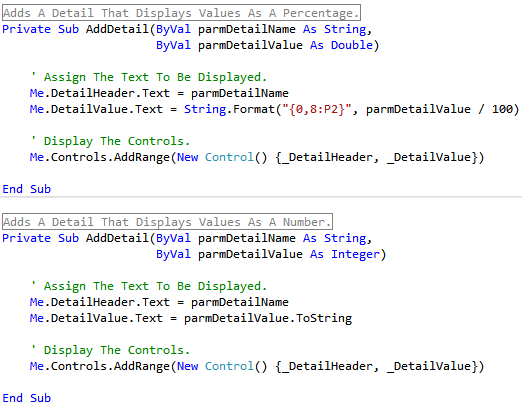
We create the following...



‘***AddDetail’*** is a method used to display the individual calculations. It has one overloaded method.

The first parameter in both methods is of type String. This represents the detail name such as the class name(8NSS) or grade type (Pass).

However, the second parameter of the first method is of type Double which represents an ‘Average Percentage’ calculation, whereas the parameter of the second method is of type Integer and represents a ‘Count’ calculation. Both parameters require different formatting.



‘*DetailHeader*’ and ‘*DetailValue*’ are both properties of type *TextBox*.

These methods and properties create the following.

The Average Percentage output...

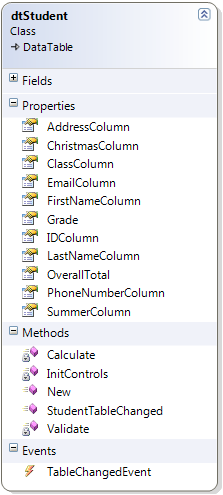


And the Count output...



# Data Objects

## The Student Data Table - *‘dtStudent.vb’*



*This component inherits from the* ***DataTable*** *class and is used to cache student records.*

We have eleven properties of type *DataColumn*, a constructor and an initialiser for adding the data columns to the table. The class also has two members for validating and calculating overall student marks and grades.

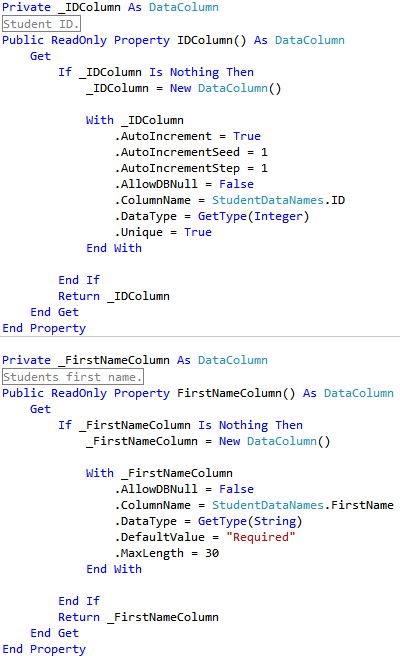
This table is the only table that is added to the student data set, *‘dsStudent.vb’*.

You may also have noticed that we have added two extra columns; *OverallTotal* and *Grade*. We’ll go into these after we explain a little about how we create a data column.

This table is passed, *By Reference*, to the student data adapter and the student totals objects.



The following examples show how Data columns are created...



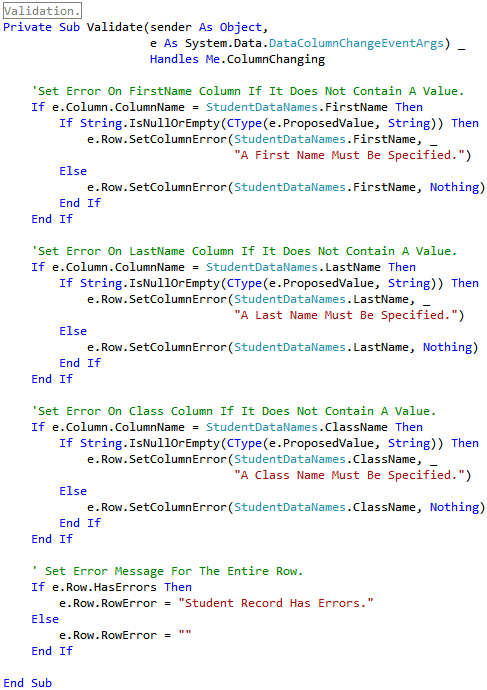
For the ‘ID’ column, you can see that we specify the auto increment seed and step because the data type for this column in the database is of type ‘AutoNumber’. We do not allow *Null* values because it is also the *Primary Key* for this table, and that is why we also set the *Unique* property to *True*. The field size in the database table is a *Long Integer*, and so we specify the data column data type to be an integer so as to handle the range of values possible for this data type.

For the *FirstName* column, we specify that *Null* values are not permitted. That is also why we set the *DefaultValue* to “Required”, this informs the user that they must enter a value. We have also specified that the maximum string length is 30.

We have also set the *DefaultValue* for *LastName* and *ClassName* to “Required”, plus The *Christmas* and *summer* default values are set to zero.

### Validation

To make sure that the end user has entered a value for the *FirstName*, *LastName* and *ClassName* columns, we use of the *ColumnChanging* event to check these columns.



We check individual columns by testing for the name of the column. We then check for errors and if found, call the *SetColumnError* method to set the error on the column. Finally we set an error on the entire row by assigning a string value to the *e.Row.RowError* property.

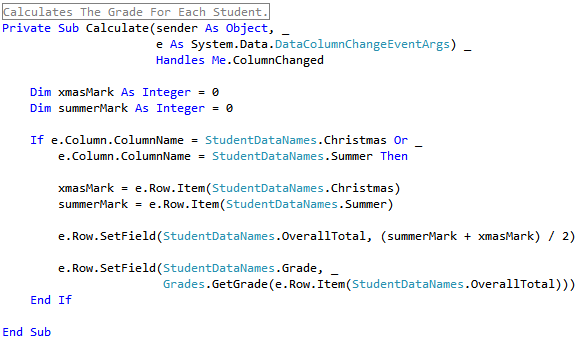
This is how errors are displayed if values for these columns are not entered..



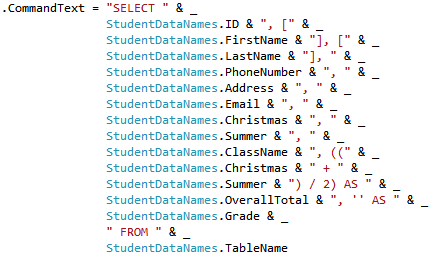
### Calculating the Overall Mark & Assigning The Grade

As mentioned earlier, we have two extra columns; *OverallTotal* and *Grade* that are not part of the database table.

We use the *ColumnChanged* event for calculating the overall grade and assigning the appropriate grade when the user is entering the information. This occurs when the end user enters a value for the *Christmas* and *Summer marks* and then *Tabs* out of these cells.



When filling the student data table, the Command*Text* property of our *SELECT* Command contains the following code for selecting all students...

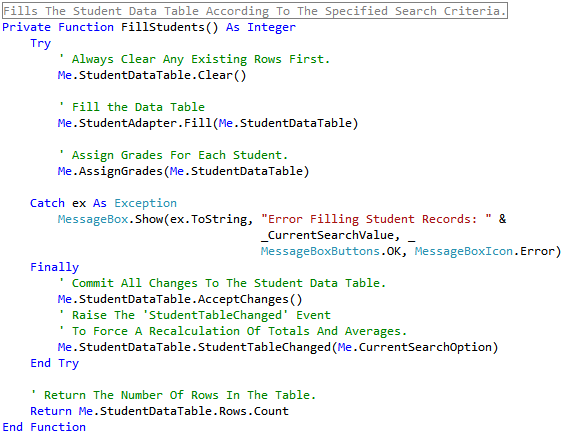


The more English version looks like the following when run...

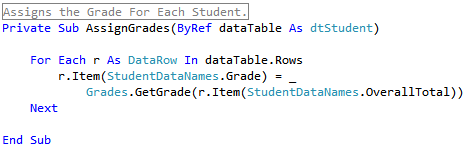


We cannot assign the grade in our SELECT statement, so, to assign the grade when filling a table, we need to call another method that iterates through the data table after it has been ***filled***, and assign the appropriate grade to each student.

When we fill the table, we call the *AssignGrades* method ...

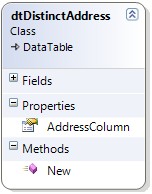


And this is how we assign the grade when called from the above method...



## Distinct Addresses - ‘*dtDistinctAddresses.vb’*

This table simply caches a collection of distinct student addresses, and used as the *DataSource* for the search value combo box, when we are filtering student records by address. The class inherits from the ***DataTable*** class and has one data column property and one method, the constructor. As you will see, all that happens is that the class adds the data column to itself.

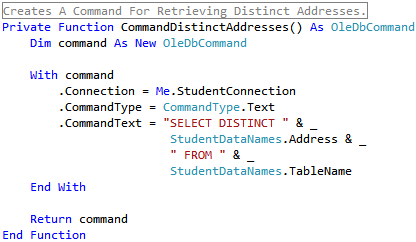


*This component also inherits from the* ***DataTable*** *class and is used to cache* ***Distinct Addresses****, used for filtering student records.*

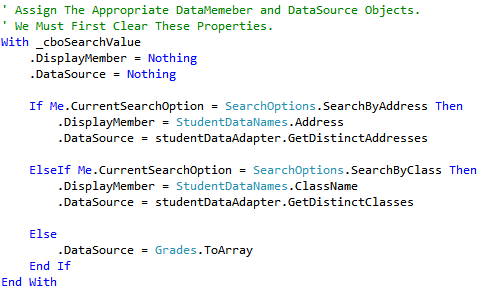
This class contains one property of type *DataColumn*, and one constructor that adds the *DataColumn* to the *DataTable*.

The following describes how we fill this table.

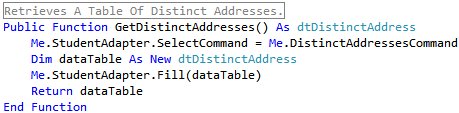
In our data adapter class, the SELECT Command for filling this table contains the DISTINCT keyword. This command searches *All* student records in the database and returns only distinct addresses...



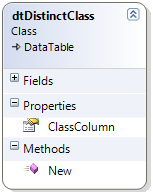
Every time the *search option* changes, the appropriate *search value* control is displayed. Whenever we want to search by address, class or grade, a combo box is used. When this happens we assign the appropriate *DataSource*.



When getting distinct addresses, we call the *GetDistinctAddressses* method which contains the following code...

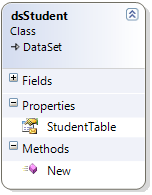


## Distinct Classes - *‘dtDistinctClasses.vb’*



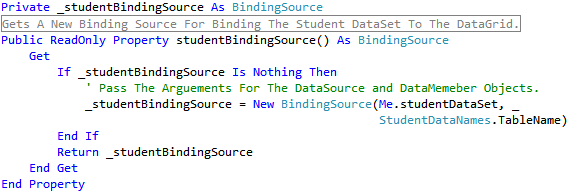
*This component works exactly like the* ***Distinct Addresses*** *class, just change the name from Address to Class.*

## The DataSet – ‘*dsStudent.vb’*

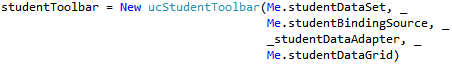


*This component inherits from the* ***DataSet*** *class and has one property that references a student Data Table object.*

When instantiated from the main form class *frmStudent.vb*, we assign it as the *DataSource* for our *BindingSource* object.

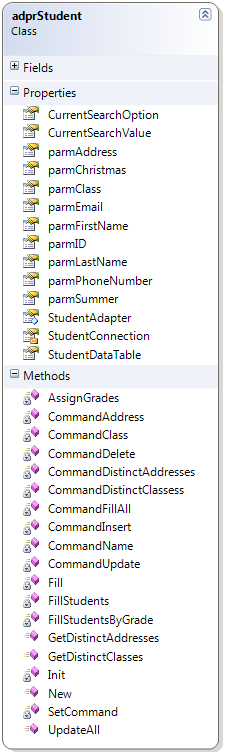


We then pass it, *By Reference* , to the Toolbar class where it used to check for changes and errors.



## The Data Adapter – *‘adprStudent.vb’*

This class handles all updates to the database and retrieves all stude**n**t records as well.

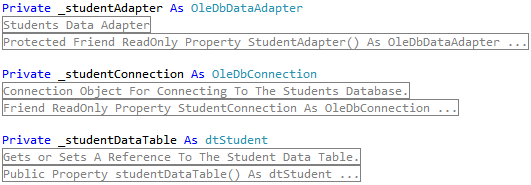


*The* ***Methods*** *section contains all the functions for creating our* ***command*** *objects, and also the methods for* ***Filling*** *our data tables.*

*In the* ***Properties*** *section, all our connection objects are declared, including an OleDbDataAdapter object and all the parameters required for connecting to the database.*

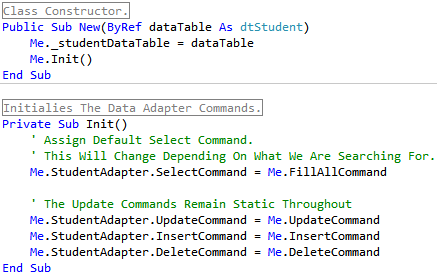
*The connection string is located in the* ***Settings*** *file.*

The adapter class contains the following connection objects...



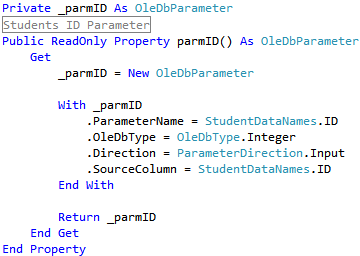
Its constructor takes a student data table as an argument and assigns it to a class property as specified above. The *Init* method assigns the default *update* and *select* command objects.

We only have a single table in the data set, that’s why we passed the data table to the constructor, rather than passing it to every ***fill*** method.



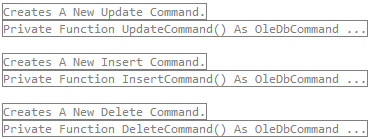
This class also contains properties that represent all the columns in the student database data table.

Here’s an example of one, the *ID* Column.

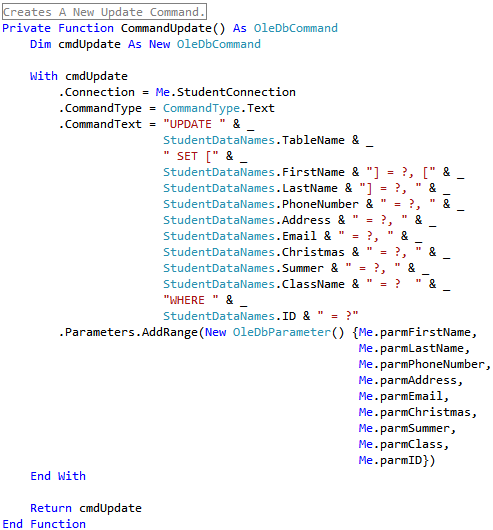


### Updating

There are three functions that create our insert, update and delete commands...

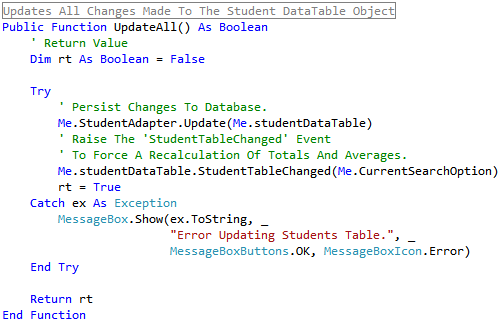


The Update Command looks like this...



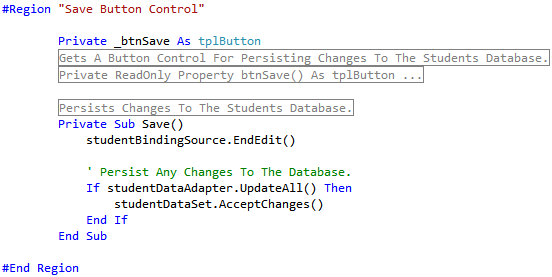
As you can see, this function returns an *OleDbCommand* object. To reduce errors, we use the static class *StudentDataNames* when naming parameters and creating our SQL statements.

The *public* method for updating the database table is as follows...



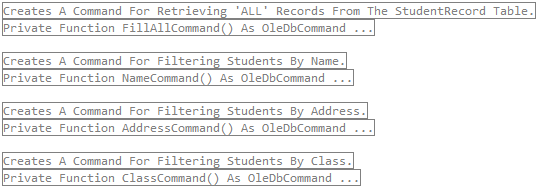
We use a *Try / Catch* structure to handle any errors. The *Update* method of the data adapter performs the actual updating, persisting all inserts, changes and deletes to the database.

In the *Toolbar* class, where the data adapter is referenced, the *Save* button calls the Update method...



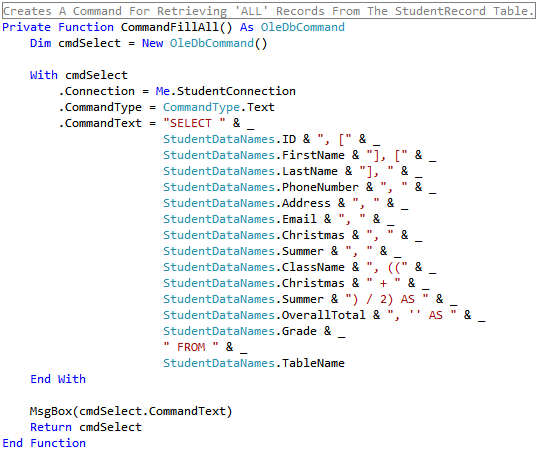
### Selecting Student Records

As we can filter student records by Name, Address, Class and Grade, we have four commands to facilitate this.

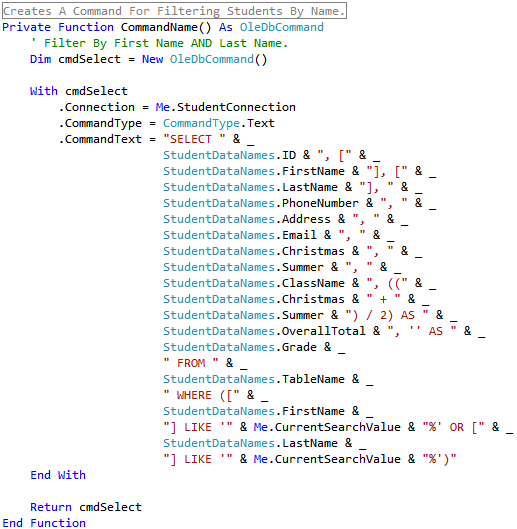


The *CommandFillAll* grabs all student records and is used when we have either no search criteria specified OR when we are searching for students that earned a particular grade. The reason for the later is that the column *Grade* is not part of the database table, so we must first get all students and then iterate through the table and filter it. The remaining three should be self explanatory.

Here’s a look at the *CommandFillAll* command*...*

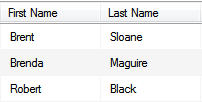


The *CommandName* command is used for searching students either by first or last name.

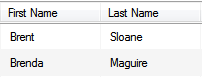


Notice how we use the keyword *LIKE* along with the *%* symbol on both the first and last name columns. The ‘%’ symbol acts as a wild card character in Access. We search both *firstname* and *lastname* columns for a given criteria, they are not treated separately.

Try it, type *‘B’* into the search box and hit enter. You’ll get the following students ...



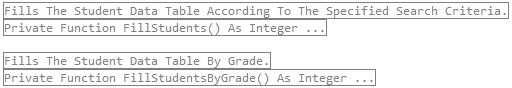
Now type ‘r’ after the *‘b* and hit enter*’*, and you’ll see the following students listed...



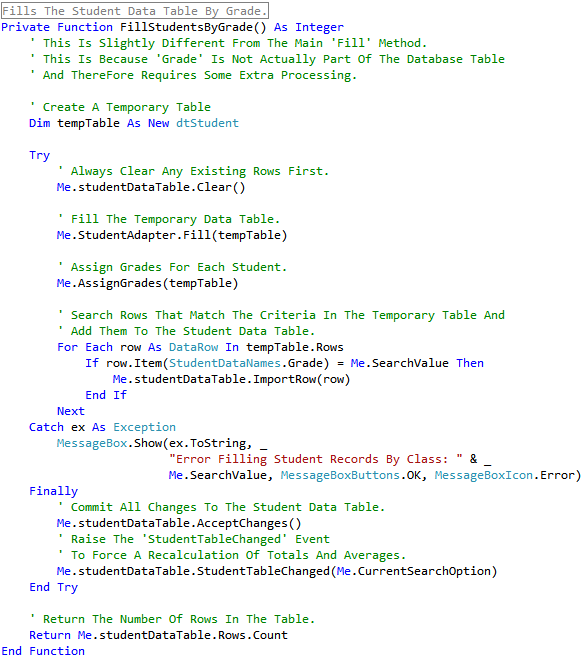
If you had typed *‘l’* instead of ‘*r’*, you would be viewing the following student (because *lastname* begins with *bl)*...



There are two methods for filling the student data table...

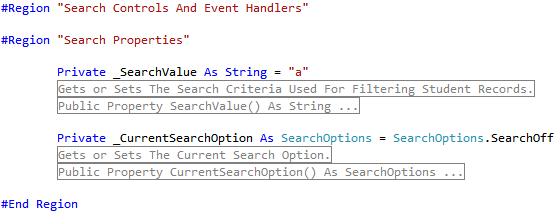


Both methods are almost identical except that the later has some extra processing to do in order to filter students by grade.



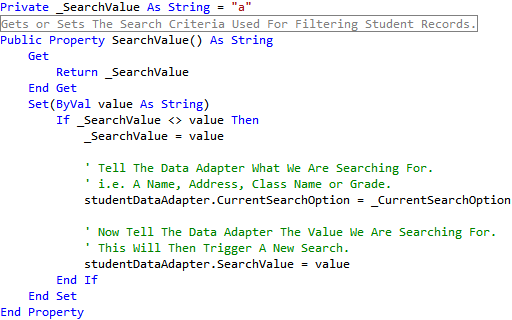
#### How do we determine what to search for ?

In the ‘*ucStudentTolbar.vb’* class, where we have our search controls, we have two properties...

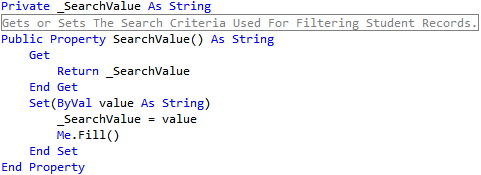


The Data Adapter class has exactly the same two properties.

When a search value is changed, we assign the new value to the \_*SearchValue* variable, through the *SET* method of its respective property, which in turn assigns the value to the same property in the data adapter.



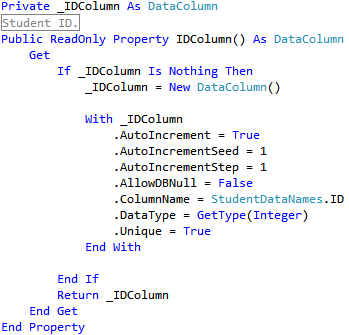
The data adapter then calls the *Fill* method.



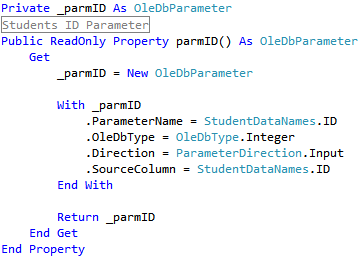
# Binding

For this, we are going to demonstrate how we bind the *ID* column in the student database table, to the *ID* column in the data grid. This should also highlight the importance of the *StudentDataNames* class.

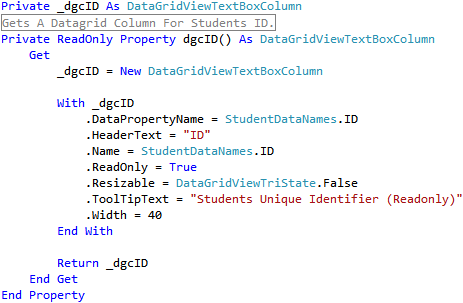
In our student data table we have a property that represents the ID column.



In our student data adapter we have the following Parameter...



Finally, a Data Grid Column in the student Data Grid...



For binding purposes, the following statements are the most important...

Data Adapter 

Data Table 

Data Grid 

The *SourceColumn* tells the adapter where to retrieve the information from in the database and the *ColumnName* in the data table, tells the adapter where to put it. Finally, the *DataPropertyName* in the data grid, tells it the name of the column in the data table to display the information from.

For all this to work we need to do three things ...

*Firstly*, we need to tell the data adapter what table to ***fill,*** by passing the data table object to its constructor.



*Secondly*, we then need to create a *BindingSource* that will tell the data grid where it’s getting its information from (we specify the dataset and the table to work from).



*Finally*, we pass the binding source to the data grids constructor which then assigns it to the *DataSource* property.



# Problems

One problem discovered at the beginning was that we could not save changes to the database table. The reason for this was because we did not have a *Primary Key*. To solve this, we simply opened the database using Access and added one, called *ID* and set its data type to *AutoNumber.*

The table now contains the following columns...

