## Lab: Adding Data Validation and Type-Safety to the Application

### Lab Setup

Estimated Time: **75 minutes**

### Exercise 1: Implementing the Teacher, Student, and Grade Structs as Classes

#### Task 1: Convert the Grades struct into a class

1. Click **Visual Studio 2017**.
2. In **Visual Studio**, on the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, browse to **E:/Allfiles/Mod04/Labfiles/Starter/Exercise 1**, click **GradesPrototype.sln**, and then click **Open**.
4. On the **View** menu, click **Task List**.
5. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1a: Convert Grade into a class and define constructors** task.
6. In the code editor, below the comment, modify the **public struct Grade** declaration, replacing **struct** with **class**.

public class Grade

1. Click at the end of the code **public string Comments { get; set; }**, press Enter twice, and then type the following code:

// Constructor to initialize the properties of a new Grade

public Grade(int studentID, string assessmentDate, string subject, string assessment, string comments) {

StudentID = studentID;

AssessmentDate = assessmentDate;

SubjectName = subject;

Assessment = assessment;

Comments = comments;

}

* // Default constructor
* public Grade() {
* StudentID = 0;
* AssessmentDate = DateTime.Now.ToString(“d”);
* SubjectName = “Math”;
* Assessment = “A”;
* Comments = String.Empty;
* }

#### Task 2: Convert the Students and Teachers structs into classes

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2a: Convert Student into a class, make the password property write-only, add the VerifyPassword method, and define constructors** task.
2. In the code editor, below the comment, modify the **public struct Student** declaration, replacing **struct** with **class**.

public class Student

1. Delete the following line of code from the **Student** class.

public string Password {get; set;}

1. Press Enter, and then type the following code:

private string \_password = Guid.NewGuid().ToString();

// Generate a random password by default

* public string Password { set { \_password = value; } }
* public bool VerifyPassword(string pass) {
* return (String.Compare(pass, \_password) == 0);
* }
* >**Note** An application should not be able to read passwords; only set them and verify that a password is correct.

1. Click at the end of the code **public string LastName { get; set; }**, press Enter twice, and then type the following code:

// Constructor to initialize the properties of a new Student

public Student(int studentID, string userName, string password, string firstName, string lastName, int teacherID) {

StudentID = studentID;

UserName = userName;

Password = password;

FirstName = firstName;

LastName = lastName;

TeacherID = teacherID;

}

* // Default constructor
* public Student() {
* StudentID = 0;
* UserName = String.Empty;
* Password = String.Empty;
* FirstName = String.Empty;
* LastName = String.Empty;
* TeacherID = 0;
* }

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2b: Convert Teacher into a class, make the password property write-only, add the VerifyPassword method, and define constructors** task.
2. In the code editor, below the comment, modify the **public struct Teacher** declaration, replacing **struct** with **class**.

public class Teacher

1. Delete the following line of code:

public string Password {get; set;},

1. Press Enter and then type the following code:

private string \_password = Guid.NewGuid().ToString();

// Generate a random password by default

* public string Password {
* set { \_password = value; }
* }
* public bool VerifyPassword(string pass) {
* return (String.Compare(pass, \_password) == 0);
* }

1. Click at the end of the code **public string Class {get; set;}**, press Enter twice, and then type the following code:

* // Constructor to initialize the properties of a new Teacher  
  publicTeacher(int teacherID, string userName, string password, string firstName,
* string lastName, string className)  
  {  
   TeacherID = teacherID;  
   UserName = userName;  
   Password = password;  
   FirstName = firstName;  
   LastName = lastName;  
   Class = className;  
  }
* // Default constructor  
  public Teacher()  
  {  
   TeacherID = 0;  
   UserName = String.Empty;  
   Password = String.Empty;  
   FirstName = String.Empty;  
   LastName = String.Empty;  
   Class = String.Empty;  
  }

#### Task 3: Use the VerifyPassword method to verify the password when a user logs in

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3a: Use the VerifyPassword method of the Teacher class to verify the teacher’s password** task.
2. In the code editor, below the comment, in the code for the teacher variable, modify the **String.Compare(t.Password, password.Password) == 0** code to look like the following code:

t.VerifyPassword(password.Password)

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3b: Check whether teacher is null before examining the UserName property** task.
2. In the code editor, in the line below the comment, modify the **if** statement condition from **!String.IsNullOrEmpty(teacher.UserName)** to look like the following code:

teacher != null && !String.IsNullOrEmpty(teacher.UserName)

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3c: Use the VerifyPassword method of the Student class to verify the student’s password** task.
2. In the code editor, below the comment, in the code for the student variable, modify the **String.Compare(s.Password, password.Password) == 0** code to look like the following code:

s.VerifyPassword(password.Password)

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3d: Check whether student is null before examining the UserName property** task.
2. In the code editor, in the line below the comment, modify the **if** statement condition from **!String.IsNullOrEmpty(student.UserName)** to look like the following code:

student != null && !String.IsNullOrEmpty(student.UserName)

#### Task 4: Build and run the application, and verify that a teacher or student can still log on

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** Menu, click **Start Without Debugging**.
3. In the **Username** text box, type **vallee**.
4. In the **Password** text box, type **password**, and then click **Log on**.
5. Verify that the welcome screen appears, displaying the list of students.
6. Click **Log off**.
7. In the **Username** text box, delete the existing contents, type **grubere**, and then click **Log on**.
8. Verify that the welcome screen appears, displaying the list of subjects and grades.
9. Click **Log off**.
10. Close the application.
11. On the **File** menu, click **Close Solution**.

**Results:** After completing this exercise, the **Teacher**, **Student**, and **Grade** structs will be implemented as classes and the **VerifyPassword** method will be called when a user logs on.

### Exercise 2: Adding Data Validation to the Grade Class

#### Task 1: Create a list of valid subject names

1. In **Visual Studio**, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:/Allfiles/Mod04/Labfiles/Starter/Exercise 2**, click **GradesPrototype.sln**, and then click **Open**.
3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1a: Define a List collection for holding the names of valid subjects** task.  
   In the code editor, in the blank line below the comment, type the following code:

public static List<string> Subjects;

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1b: Populate the list of valid subjects with sample data** task.
2. In the code editor, in the blank line below the comment, type the following code:

Subjects = new List<string>() {

"Math", "English", "History", "Geography", "Science"

};

#### Task 2: Add validation logic to the Grade class to check the data entered by the user

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Add validation to the AssessmentDate property** task.
2. In the code editor under comment, delete the **public string AssessmentDate { get; set; }** code, and then type the following code:

private string \_assessmentDate;

* public string AssessmentDate {
* get { return \_assessmentDate; }
* set {
* DateTime assessmentDate;  
   // Verify that the user has provided a valid date  
   if (DateTime.TryParse(value, out assessmentDate)) {
* // Check that the date is no later than the current date  
   if (assessmentDate > DateTime.Now) {  
   // Throw an ArgumentOutOfRangeException if the date is
* // after the current date  
   throw new ArgumentOutOfRangeException("AssessmentDate",
* "Assessment date must be on or before the current date");  
   }
* // If the date is valid, then save it in the appropriate format  
   \_assessmentDate = assessmentDate.ToString("d");  
   }   
   else  
   {  
   // If the date is not in a valid format then throw
* // an ArgumentException  
   throw new ArgumentException("AssessmentDate",
* "Assessment date is not recognized");  
   }  
   }
* }

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Add validation to the SubjectName property** task.
2. In the code editor, below the comment, delete the **public string SubjectName { get; set; }** code, and then type the following code:

private string \_subjectName;

* public string SubjectName {
* get { return \_subjectName; }
* set  
   {  
   // Check that the specified subject is valid  
   if (DataSource.Subjects.Contains(value))  
   {  
   // If the subject is valid store the subject name  
   \_subjectName = value;  
   }  
   else  
   {  
   // If the subject is not valid then throw an ArgumentException  
   throw new ArgumentException("SubjectName", "Subject is not recognized");  
   }  
   }
* }

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2c: Add validation to the Assessment property** task.
2. In the code editor, delete the **public string Assessment { get; set; }** code, and then type the following code:

private string \_assessment;

* public string Assessment {
* get { return \_assessment; }
* set {  
   // Verify that the grade is in the range A+ to E-  
   // Use a regular expression: a single character in the range A-E at
* // the start of the string followed by an optional + or – at the end
* //of the string
* Match matchGrade = Regex.Match(value, @"[A-E][+-]?$");  
   if (matchGrade.Success) {  
   \_assessment = value;  
   }  
   else {  
   // If the grade is not valid then throw an ArgumentOutOfRangeException  
   throw new ArgumentOutOfRangeException("Assessment",
* "Assessment grade must be in the range of A+ to E-");  
   }  
   }
* }

#### Task 3: Add a unit test to verify that the validations defined for the Grade class functions as expected

1. On the **File** menu, point to **Add**, and then click **New Project**.
2. In the **Add New Project** dialog box, in the **Installed** templates list, expand **Visual C#**, click **Test**, and then in the **Templates** list, click **Unit Test Project (.NET Framework)**.

>**Note:** In the **Add New Project** window, make sure **.NET Framework 4.7** is selected at the bottom of the page..

1. In the **Name** text box, type **GradesTest**, and then click **OK**.
2. In **Solution Explorer**, right-click **GradesTest**, and click **Add** then click **Reference**.
3. In the **Reference Manager – GradesTest** dialog box, expand **Projects** and then click **Solution**.
4. Select the **GradesPrototype** check box, and then click **OK**.
5. In the code editor, in the **UnitTest1** class, delete all of the existing code within the class itself, and then type the following code:

[TestInitialize]

public void Init() {

// Create the data source (needed to populate the Subjects collection)

GradesPrototype.Data.DataSource.CreateData();

}

* [TestMethod]
* public void TestValidGrade() {
* GradesPrototype.Data.Grade grade = new GradesPrototype.Data.Grade(1,
* “1/1/2012”, “Math”, “A-”, “Very good”);
* Assert.AreEqual(grade.AssessmentDate, “1/1/2012”);
* Assert.AreEqual(grade.SubjectName, “Math”);
* Assert.AreEqual(grade.Assessment, “A-”);
* }
* [TestMethod]
* [ExpectedException(typeof(ArgumentOutOfRangeException))]
* public void TestBadDate() {
* // Attempt to create a grade with a date in the future
* GradesPrototype.Data.Grade grade = new GradesPrototype.Data.Grade(1,
* “1/1/2023”, “Math”, “A-”, “Very good”);
* }
* [TestMethod]
* [ExpectedException(typeof(ArgumentException))]
* public void TestDateNotRecognized () {
* // Attempt to create a grade with an unrecognized date
* GradesPrototype.Data.Grade grade = new GradesPrototype.Data.Grade(1,
* “13/13/2012”, “Math”, “A-”, “Very good”);
* }
* [TestMethod]
* [ExpectedException(typeof(ArgumentOutOfRangeException))]
* public void TestBadAssessment() {
* // Attempt to create a grade with an assessment outside the range A+ to E-
* GradesPrototype.Data.Grade grade = new GradesPrototype.Data.Grade(1,
* “1/1/2012”, “Math”, “F-”, “Terrible”);
* }
* [TestMethod]
* [ExpectedException(typeof(ArgumentException))]
* public void TestBadSubject() {
* // Attempt to create a grade with an unrecognized subject
* GradesPrototype.Data.Grade grade = new GradesPrototype.Data.Grade(1,
* “1/1/2012”, “French”, “B-”, “OK”);

}

1. On the **Build** menu, click **Build Solution**.
2. On the **Test** menu, point to **Run**, and then click **All Tests**.
3. In the **Test Explorer** window, verify that all the tests are passed.
4. Close **Test Explorer**.
5. On the **File** menu, click **Close Solution**.

**Results:** After completing this exercise, the **Grade** class will contain validation logic.

### Exercise 3: Displaying Students in Name Order

#### Task 1: Run the application and verify that the students are not displayed in any specific order when logged on as a teacher

1. In **Visual Studio**, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:/Allfiles/Mod04/Labfiles/Starter/Exercise 3**, click **GradesPrototype.sln**, and then click **Open**.
3. On the **Build** menu, click **Build Solution**.
4. On the **Debug** Menu, click **Start Without Debugging**.
5. In the **Username** text box, type **vallee**.
6. In the **Password** text box, type **password**, and then click **Log on**.
7. Verify that the students are not displayed in any specific order.
8. Close the application.

#### Task 2: Implement the IComparable<Student> interface to enable comparison of students

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2a: Specify that the Student class implements the IComparable<Student> interface** task.
2. In the code editor, click at the end of the **public class Student** declaration, and then type the following code:

: IComparable<Student>

**Note**: Ignore the red squiggly line at this point.

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2b: Compare Student objects based on their LastName and FirstName properties** task.
2. In the code editor, in the blank line below the comment, type the following code:

// Compare Student objects based on their LastName and FirstName properties

public int CompareTo(Student other) {

// Concatenate the LastName and FirstName of this student

string thisStudentsFullName = LastName + FirstName;

* // Concatenate the LastName and FirstName of the "other" student
* string otherStudentsFullName = other.LastName + other.FirstName;  
   // Use String.Compare to compare the concatenated names and return the result
* return(String.Compare(thisStudentsFullName, otherStudentsFullName));
* }

#### Task 3: Change the Students ArrayList collection into a List<Student> collection

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3a: Change the Students collection into a List<Student>** task.
2. In the code editor, below the comment, modify the **public static ArrayList Students;** code to look like the following code:

public static List<Student> Students;

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3b: Populate the List<Student> collection** task.
2. In the code editor, below the comment, modify the **Students = new ArrayList()** code to look like the following code:

Students = new List<Student>()

#### Task 4: Sort the data in the Students collection

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 4a: Sort the data in the Students collection** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

DataSource.Students.Sort();

#### Task 5: Verify that Students are retrieved and displayed in the order of their first name and last name

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In the **Username** text box, type **vallee**.
4. In the **Password** text box, type **password**, and then click **Log on**.
5. Verify that the students are displayed in the alphabetical order, based on their last names.
6. Log off and then close the application.
7. On the **File** menu, click **Close Solution**.

**Results:** After completing this exercise, the application will display the students in the alphabetical order, based on last names, and then first names.

### Exercise 4: Enabling Teachers to Modify Class and Grade Data

#### Task 1: Change the Teachers and Grades collections to be generic List collections

1. In **Visual Studio**, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:/Allfiles/Mod04/Labfiles/Starter/Exercise 4**, click **GradesPrototype.sln**, and then click **Open**.
3. In the **Task List** window, double-click the **TODO: Exercise 4: Task 1a: Change the Teachers collection into a generic List** task.
4. In the code editor, below the comment, modify the code **public static ArrayList Teachers;** to look like the following code:

public static List<Teacher> Teachers;

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 1b: Change the Grades collection into a generic List** task.
2. In the code editor, below the comment, modify the code **public static ArrayList Grades;** to look like the following code:

public static List<Grade> Grades;

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 1c: Populate the Teachers collection** task.
2. In the code editor, below the comment, modify the code **Teachers = new ArrayList()** to look like the following code:

Teachers = new List<Teacher>()

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 1d: Populate the Grades collection** task.
2. In the code editor, below the comment, modify the code **Grades = new ArrayList()** to look like the following code:

* Grades = newList<Grade>()

#### Task 2: Add the EnrollInClass and RemoveFromClass methods for the Teacher class

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2a: Enroll a student in the class for this teacher** task.
2. In the code editor, click in the blank line below the comment, and then type the following code:

public void EnrollInClass(Student student) {

// Verify that the student is not already enrolled in another class

if (student.TeacherID == 0)

{

// Set the TeacherID property of the student

student.TeacherID = TeacherID;

}

else

{

// If the student is already assigned to a class, throw an ArgumentException

throw new ArgumentException("Student",

"Student is already assigned to a class");

}

}

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2b: Remove a student from the class for this teacher** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

// Remove a student from the class for this teacher

public void RemoveFromClass(Student student) {

// Verify that the student is actually assigned to the class for this teacher

if (student.TeacherID == TeacherID) {

// Reset the TeacherID property of the student

student.TeacherID = 0;

}

else

{

// If the student is not assigned to the class for this teacher,

// throw an ArgumentException

throw new ArgumentException("Student",

"Student is not assigned to this class");

}

}

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2c: Add a grade to a student (the grade is already populated)** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

// Add a grade to a student (the grade is already populated)

public void AddGrade(Grade grade) {

// Verify that the grade does not belong to another student

// – the StudentID should be zero

if (grade.StudentID == 0)

{

// Add the grade to the student’s record

grade.StudentID = StudentID;

}

else

{

// If the grade belongs to a different student, throw an ArgumentException

throw new ArgumentException("Grade",

"Grade belongs to a different student");

}

}

#### Task 3: Add code to enroll a student in a teacher’s class

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 3a: Enroll a student in the teacher’s class** task.
2. In the code editor, below the comment, click in the blank line in the **Student\_Click** method, and then type the following code:

try

{

// Determine which student the user clicked

// the StudentID is held in the Tag property of the Button that the user clicked

Button studentClicked = sender as Button;

int studentID = (int)studentClicked.Tag;

* // Find this student in the Students collection
* Student student = (from s in DataSource.Students
* where s.StudentID == studentID
* select s).First());  
   // Prompt the user to confirm that they wish to add this student to their class
* string message = String.Format("Add {0} {1} to your class?",
* student.FirstName, student.LastName);
* MessageBoxResult reply = MessageBox.Show(message, "Confirm",
* MessageBoxButton.YesNo, MessageBoxImage.Question);
* if (reply == MessageBoxResult.Yes)
* {
* // Get the ID of the currently logged-on teacher
* int teacherID = SessionContext.CurrentTeacher.TeacherID;
* // Assign the student to this teacher’s class
* SessionContext.CurrentTeacher.EnrollInClass(student);
* // Refresh the display – the new assigned student should disappear from
* // the list of unassigned students
* Refresh();
* }
* }
* catch (Exception ex)
* {
* MessageBox.Show(ex.Message, “Error enrolling student”,
* MessageBoxButton.OK, MessageBoxImage.Error);
* }

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 3b: Refresh the display of unassigned students** task.
2. In the code editor, below the comment, click in the blank line in the **Refresh** method, and then type the following code:

// Find all unassigned students - they have a TeacherID of 0

var unassignedStudents = (from s in DataSource.Students

where s.TeacherID == 0

select s;);

* // If there are no unassigned students, then display the
* // “No unassigned students” message
* // and hide the list of unassigned students
* if (unassignedStudents.Count() == 0)
* {
* txtMessage.Visibility = Visibility.Visible;
* list.Visibility = Visibility.Collapsed;
* }
* else
* {
* // If there are unassigned students, hide the “No unassigned students” message
* // and display the list of unassigned students
* txtMessage.Visibility = Visibility.Collapsed;
* list.Visibility = Visibility.Visible;
* // Bind the ItemControl on the dialog to the list of unassigned students
* // The names of the students will appear in the ItemsControl on the dialog
* list.ItemsSource = unassignedStudents;
* }

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 3c: Enroll a student in the teacher’s class** task.
2. In the code editor, below the comment, click in the blank line in the **EnrollStudent\_Click** method, and then type the following code:

// Use the AssignStudentDialog to display unassigned students and

// add them to the teacher’s class

// All of the work is performed in the code behind the dialog

AssignStudentDialog asd = new AssignStudentDialog();

asd.ShowDialog();

* // Refresh the display to show any newly enrolled students
* Refresh();

#### Task 4: Add code to enable a teacher to remove the student from the assigned class

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 4a: Enable a teacher to remove a student from a class** task.
2. In the code editor, below the comment, click in the blank line in the **Remove\_Click** method, and then type the following code:

// If the user is not a teacher, do nothing (the button should not appear anyway)

if (SessionContext.UserRole != Role.Teacher)

{

return;

}

* try
* {
* // If the user is a teacher, ask the user to confirm that this student
* //should be removed from their class
* string message = String.Format(“Remove {0} {1}”,
* SessionContext.CurrentStudent.FirstName,
* SessionContext.CurrentStudent.LastName);
* MessageBoxResult reply = MessageBox.Show(message, “Confirm”,
* MessageBoxButton.YesNo, MessageBoxImage.Question);
* // If the user confirms, then call the RemoveFromClass method of the
* // current teacher to remove this student from their class
* if (reply == MessageBoxResult.Yes)
* {
* SessionContext.CurrentTeacher.RemoveFromClass(SessionContext.CurrentStudent);
* // Go back to the previous page – the student is no longer a member of the
* // class for the current teacher  
   if (Back != null)  
   {  
   Back(sender, e);  
   }  
   }
* }
* catch (Exception ex)
* {
* MessageBox.Show(ex.Message, “Error removing student from class”,
* MessageBoxButton.OK, MessageBoxImage.Error);
* }

#### Task 5: Add code to enable a teacher to add a grade to a student

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 5a: Enable a teacher to add a grade to a student** task.
2. In the code editor, below the comment, click in the blank line in the **AddGrade\_Click** method, and then type the following code:

// If the user is not a teacher, do nothing (the button should not appear anyway)  
 if (SessionContext.UserRole != Role.Teacher)  
 {  
 return;  
 }  
 try  
 {  
 // Use the GradeDialog to get the details of the assessment grade  
 GradeDialog gd = new GradeDialog();  
  
 // Display the form and get the details of the new grade  
 if (gd.ShowDialog().Value)  
 {  
 // When the user closes the form, retrieve the details of

// the assessment grade from the form  
 // and use them to create a new Grade object  
 Grade newGrade = new Grade();  
 newGrade.AssessmentDate = gd.assessmentDate.SelectedDate.Value.ToString ("d");  
 newGrade.SubjectName = gd.subject.SelectedValue.ToString();  
 newGrade.Assessment = gd.assessmentGrade.Text;  
 newGrade.Comments = gd.comments.Text;

// Save the grade to the list of grades  
 DataSource.Grades.Add(newGrade);  
  
 // Add the grade to the current student  
 SessionContext.CurrentStudent.AddGrade(newGrade);

// Refresh the display so that the new grade appears  
 Refresh();  
 }  
 }  
 catch (Exception ex)  
 {  
 MessageBox.Show(ex.Message, "Error adding assessment grade",

MessageBoxButton.OK, MessageBoxImage.Error);  
 }

#### Task 6: Run the application and verify that students can be added to and removed from classes, and that grades can be added to students

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In the **Username** text box, type **vallee**.
4. In the **Password** text box, type **password**, and then click **Log on**.
5. Click **New Student**.
6. In the **First Name** text box, type **Darren**.
7. In the **Last Name** text box, type **Parker**.
8. In the **Password** text box, type **password**, and then click **OK**.
9. Click **Enroll Student**.
10. Verify that the **Assign Student** dialog box appears and that **Darren Parker** is in the list.
11. Click **Darren Parker**.
12. Verify that the **Confirm** message box appears, and then click **Yes**.
13. In the **Assign Student** dialog box, verify that **Darren Parker** disappears and that the text **No unassigned students** is displayed.
14. Click **Close**.
15. Verify that **Darren Parker** is added to the student list.
16. Click the student **Kevin Liu**.
17. Click **Remove Student**.
18. Verify that the **Confirm** message box appears, and then click **Yes**.
19. Verify that **Kevin Liu** is removed from the student list.
20. Click the student **Darren Parker**.
21. Click **Add Grade**.
22. Verify that the **New Grade Details** dialog box appears.
23. Verify that the **Date** text box contains the current date.
24. In the **Subject** list, click **English**.
25. In the **Assessment** text box, type **B**.
26. In the **Comments** text box, type **Good**, and then click **OK**.
27. Verify that the grade information appears on the report card.
28. Click **Log off**.
29. In the **Username** text box, type **parkerd**.
30. Click **Log on**.
31. Verify that the **Welcome Darren Parker** screen is displayed, showing the report card and the previously added grade.
32. Click **Log off**.
33. Close the application.
34. In **Visual Studio**, on the **File** menu, click **Close Solution**.

**Results:** After completing this exercise, the application will enable teachers to add and remove students from their classes, and to add grades to students.