|  |  |  |
| --- | --- | --- |
| Name: | Cameron VanHouzen | |
|  | | |
| Grade: |  | of 50 |

# Instructions

Create a new folder named “<nmcid>-final” using your own NMCID (example: *schmoej-final*). Use this folder for all exam files, including a copy of this Word document containing your name and your answers.

# (38) 3-Tier Application – Music

In this problem you will create new C# application ***Music*** to display info about orchestral instruments in database Orchestra.accdb. Please read through the entire problem description before you begin.

**Important note! Several independent C# programming concepts have been combined into this problem. If you are unable to finish one part, make sure you move ahead to implement other parts to earn their associated points.** (Examples of partial credit solutions: replace the enum with strings; pull only one field instead of two from the database; implement one of the two GetInstruments() methods.) **Make sure your completed application compiles and runs. If you can’t get a feature to work, disable it with comments and describe your problem with additional comments in your code.**

**Application configuration**

Copy database Orchestra.accdb downloaded from Moodle. Choose a relative path (best) or a fixed path (E:\Data\ or equivalent). Open the database in Access to become familiar with its schema and data. Alternative: Create and use a SQL Server database from the Access file or the XML file.

(1 point) Create application setting orchestraConnection defining the provider and data source for the Orchestra database.

(1 point) Create public enum Section containing four values – String, Woodwind, Brass, and Percussion. Place the definition in its own file. Use standard file naming conventions.

(1 point) Add XML documentation for the enum and for each value within the enum – five places. Add identifying documentation at the top of the file, identifying the purpose of the enum, the programmer, and the date.

**Public individual “business” class Instrument**

Each instance of this class will correspond to a single record from the *Instruments* table.

(1 point) Create new custom class Instrument.

(1 point) Add string property Name to hold the name of a musical instrument (ex: “Trombone”). Add a corresponding private field. Trim leading and trailing white space.

(1 point) Enforce Title Case.

(1 point) Use a regular expression to ensure the value contains only letters and spaces. Throw an applicable Exception if rules fail.

(1 point) Add auto-generated *Section[[1]](#footnote-1)* property Category to hold the classification of the musical instrument (ex: *Section.Brass*).

(1 point) Add a fully-specified constructor that takes two parameters: instrument’s name and category.

(1 point) Add a default constructor. Using two constants for default values, call the fully-specified constructor from this one.

(1 point) Override the definition of ToString().

(1 point) Add application documentation at the top of the file, identifying the purpose of the class, the assignment, the programmer, and the date. Add several applicable comments to document your code.

(1 point) Add XML documentation for the *Instrument* class, each property, each constructor, and the method – six total places in this file.

**Data manipulation class DataRepository**

This class encapsulates all database operations, separating the user interface from backend database-specific commands. Methods in this class return business data to the user interface class.

(1 point) In new custom class DataRepository, add appropriate using statement(s) for working with databases (polymorphically) and for using application settings.

(1 point) Create a DbConnection class-level variable to refer to the database. Use the connection string parameter constructor, obtaining the value from orchestraConnection.

(1 point) Create a DbCommand class-level variable and a DbDataReader class-level variable.

(1 point) Explain similarities and distinctions between generic ADO data provider objects such as these and their OleDb or Sql equivalents:

|  |  |
| --- | --- |
| Similarities | Sql data provider descends from ADO data provider and must be able to accomplish anything that an ADO data provider can |
| Distinctions | Sql data provider may have functions that are not available to an ADO Data provider |

Add method GetInstruments(Section selectedSection) that returns a List<Instrument> collection.

(1 point) Correctly configure the DbCommand.

(1 point) Use a correct SQL statement to select only those records that match selectedSection, sorting results in alphabetic order by Instrument. Recommended: use an SQL parameter. Optional: use LINQ.

(1 point) Create a new *Instrument* instance from each record returned by the DbDataReader using the fully-specified constructor. Add each to a local List<Instruments> variable.

(1 point) Correctly cast the Category value as a member of enum Section.

(1 point) Surround all risky database code with structured exception handling. Optional – and a good idea – also manage the iDisposable DbConnection object with a *using* statement.

Add overloaded method GetInstruments() that also returns a List<Instrument> collection. *Optional: you are welcome to implement this method by calling GetInstruments(Section selectedSection).*

(1 point) Correctly configure the DbCommand.

(1 point) Use SQL to select all records and to sort in ascending order by Instrument. Optional: use LINQ to sort.

(1 point) Create a new *Instrument* instance from each record returned by the DbDataReader using the fully-specified constructor. Add each to a local List<Instruments> variable.

(1 point) Correctly cast the Category value as a member of enum Section.

(1 point) Surround all risky database code with structured exception handling. Optional – and a good idea – also manage the iDisposable DbConnection object with a *using* statement.

Complete the DataRepository class:

(1 point) Add application documentation at the top of the file: identify the purpose of the class, the assignment, the programmer, and the date. Add several applicable comments to document your code. Add XML documentation for the *DataRepository* class and for each method – three places.

You are welcome to add other features if you find them helpful and appropriate. However, no other features are required in this class and none will be graded.

**User interface class**

Instantiate a *DataRepository* object.

(2 points) Add a feature that displays the results of the *GetInstruments*() method on the form. Explain your approach:

|  |
| --- |
| Universal checkbox group, checking all types would be the equivalent of GetInstruments() |

(2 points) Add a feature that displays the results of the *GetInstruments*(*Section* *selectedSection*) method using a user-specified *Section*. Explain your approach:

|  |
| --- |
| Universal checkbox group, checking one type would be the equivalent of GetInstruments() |

(2 points) Implement professional-quality style in your graphical user interface: appropriate anchoring, size management, tab order, color, font, graphics, labels, etc. List four specific and distinct design features you would like me to review for grading:

|  |  |
| --- | --- |
| Check all and uncheck buttons | Query Counter |
| Appropriate Icon | Calibri!!!! |

(1 point) Add application documentation at the top of the file, identifying the purpose of the GUI, the assignment, the programmer, and the date. Add several applicable comments to document your code. Add XML documentation for the user interface class and for each public member.

**Wrap-up**

(1 point) Remove unused *usings* in each file. Add region blocks around business rules, fields and class-level variables, properties, and methods as applicable. Collapse all regions. Configure an XML documentation file for your project. Rebuild your solution and resolve XML warnings, if applicable.

(2 points) Draw a class diagram showing all classes and their relationships. Paste a graphic image here.

|  |
| --- |
| VS2013 has this really nice class diagram builder check it |

**Going further with the user interface – optional**

(1 point extra credit) Use the enum’s values (rather than design-time entered strings) to populate the choice of *selectedSection* on the GUI. For example, if you use CheckBoxes, set the Text property of the first CheckBox to be “*String*”, the second to be “*Woodwind*”, etc., by reading values from enum *Section* in the form’s constructor or in a form load event.

(1 point extra credit) Add a feature that lets the user select multiple *Sections* (such as *Brass* and *Woodwind*) at the same time. Explain your approach:

|  |
| --- |
| Used a generated list of all enums as checkboxes with an event that fires a query when any change and at least one is selected |

# (12) Object-Oriented Programming Definitions

Use your own words in a concise, complete definition of each of these terms. List a situation / example for each.

(2 points) Interface

|  |  |
| --- | --- |
| Definition | A set of undefined properties and methods that can be implemented by a class and allow for a high level of abstraction |
| Example | A Method that takes IMotorVehicle as a parameter could take a Mustang GT as an argument since the Mustang implements the IMotorVehicle interface |

(2 points) Generic collection

|  |  |
| --- | --- |
| Definition | Strongly typed collections which allow users to create collections that can be manipulated easily since they can all assumed to be the same type |
| Example | A List<Cars> is a strongly typed collection of cars that can have the Car properties of it accessed through the collection since they all have to be Cars |

(2 points) Namespace

|  |  |
| --- | --- |
| Definition | Namespaces are used to organize all or a subset of classes in a project, all classes must fall under a namespace. They can be referenced in other namespaces to access their classes |
| Example | Namespace Car contains some classes in my VehicleProject solution and in my Airplane namespace I can add a reference to it to access all of my classes from CarProject |

(2 points) Class library

|  |  |
| --- | --- |
| Definition | A solution that is compiled into a dll file that can be referenced in other projects |
| Example | I can turn CarProject into a dll file and in my AirplaneProject Solution I can add a reference to the CarProject.dll to access the classes in it |

(2 points) Nested class

|  |  |
| --- | --- |
| Definition | A class that is declared inside of another class, to reference it from outside first the parent class must be referenced. Often used when a class is dependant on the parent class |
| Example | If I have Class Engine and class Piston it might make sense to nest Piston inside of Engine |

(2 points) Click-Once deployment

|  |  |
| --- | --- |
| Definition | A deployment method available to .NET applications that allows for rapid deployment of applications through a click once executable |
| Example | When I’m finished with CarProject I can create a click-once installer for it that I can distribute to my clients. |

# Final Instructions

Type your name below to attest that you did not interact with other students or experts via any means during this exam, that you did not borrow solutions or hints from another student or from any prior student(s), and that you did not and will not share your solutions or any hints with other students – current or future:

|  |
| --- |
| Cameron VanHouzen |

Make sure you’ve saved your completed C# application and this document in your final exam folder (the one named “<*your NMCID*>-final”). Compress the folder and submit it via Moodle.

1. Optional: if you prefer, you can substitute the nullable *Section?* instead of *Section* in this application. If you do so, you may find that you need to include a corresponding private field. [↑](#footnote-ref-1)