PROJECT REQUIREMENTS

The students have to create a project work on their own. The project has to be designed according to the layering principles of the semester, and also all the technologies have to be used that they learn throughout the semester. Everyone has to create a project work, even those who already completed a part of the subject.

The created program has to implement some business tasks using a database that contains minimum three inter-connected tables.

During the preparation of the project, the following expectations must be met:

- Dotnet 5.0.x projects
- DoxyGen generated HTML/CHM/PDF documentation
- The code must be StyleCop+NetAnalyzers validated, "zero warnings"
- Usage of a database + Entity Framework (Code-First/Sql-First) to access it
- Usage of LINQ
- Code with unit tests (typically for the business logic classes)
- Layered architecture (Business operations and data logic must be in separate layers from the UI, so typically minimum 5 projects:
  Console App + Business logic + Repository + Data + Tests)
- Single-user, single-branch GIT repository

For the project, it is necessary to create a GIT repository at bitbucket.org, using the following naming convention: OENIK_PROG3_YEAR_SEMESTER_NEPTUN, where YEAR is the year when the semester starts; SEMESTER means **1 = spring, 2 = autumn**. In addition to the team members, admin access must be given to the project repository to the bitbucket user **oe_nik_prog** (this user is accessible by all teachers, and its e-mail address is: **nikprog@iar.nik.uni-obuda.hu**).

The file **prog_tools_en.pdf** contains a short description about the tools that should be used throughout the semester, such as Git, Doxygen, FxCop/StyleCop, etc...

Additional materials:

- prog_tools_en.pdf ⇒ https://users.nik.uni-obuda.hu/prog3/
- official subject requirements document ⇒ https://users.nik.uni-obuda.hu/prog3/
- NikGitStats website ⇒ https://users.nik.uni-obuda.hu/gitstats/

PROJECT SPECIFICATION

The students have to create a project work on their own. The project has to be designed according to the layering principles of the semester, and also all the technologies have to be used that they learn throughout the semester. Everyone has to create a project work, even those who already completed a part of the subject.

It is required for the project exercise to create a database using the code-first/sql-first approach. The database must have minimum 3 tables, that are referencing each other using foreign keys. Every table must contain minimum 5 non-key fields. If you use a connector table, that does not count into the 3 required tables.

The task: full management of these three tables (list + add + modify + remove), and a couple of (minimum 3) additional functions that do more than simply list a single table, where a **table join (not lazy load) AND group by** is required to get the desired output. The modifications don't have to be generic/flexible, but all 3 entities must be modifiable to some degree.

Example: **carBrands** (id, name, countryName, url, foundation year, yearly revenue) + **models** (id, brand_id, name, date of arrival, engine volume, horse power, base price) + **extra features** (id, category name, name, price, color, is_multiple_allowed) + **modellExtraConnector** (id, modell_id, extra_id).

Example functions:

- List / add / modify / remove brands
- List / add / modify / remove models
- List / add / modify / remove extra features
- List / add / remove modell-extra feature associations
- For all cars, we must write out the FULL price: base price + sum price of all extras on the car
- For all brands, we want to write out the average base price of the cars
- For all extra feature category names, we want to write out the number of usages on the car modells

In the following page you can find the schedule of the project. The CRUD abbreviation stands for the Create,Read,Update,Delete functions, so the basic read/write functionalities.

Usage of this car example is **NOT** allowed: please find out some simple structure on your own. It is allowed (and advised) to use the same table structure as with the databases project work. The developed project work must have similar features as the ones listed above (all tables should have all CRUD features, plus three extra non-CRUD more complex features).

At the end of the semester students must hand in the full and correct source code via Git; also via Git the Doxygen developer documentation; and via Moodle a short, 1-2 minutes

long video in which the student demonstrates the usage and the operation of all menuitems of the application.

Date	**Name**	**Must be ready with…**
06/MAR 23:55	Creation of the git repo	- Bitbucket registration, new repo (use the name OENIK_PROG3_YEAR_SEMESTER_NEPTUN), oe_nik_prog ⇒ admin permission, SourceTree install - Find out a nice project name (no spaces/accents, e.g. MySongShop); find out a nice topic and functions - .gitignore and readme.md files in the repository root (readme.md contents: project name, list of functions for the console app) - .gitignore must be using the contents from VisualStudio.gitignore, SQL Server entries must be commented out - The files can be edited using the Bitbucket website
20/MAR 23:55	Solution, all projects	The project names must use the title of the project that you established in step 1 - the solution name MUST be the same as the repo name Create the projects within the git repository (Dotnet Core projects) - MySongShop.Data - Class Library - MySongShop.Repository - Class Library - MySongShop.Logic - Class Library - MySongShop.Logic.Tests - Class Library - MySongShop.Program - Console App NuGet/Add packages to projects - All projects should have StyleCop and FxCopAnalyzers added - *.Test should have nUnit (v3), nUnit3TestAdapter, Moq - Add ConsoleMenu-simple to the Program **Layering rules:** - The Console App can only call Logic operations, the logic forwards the CRUD operations to the Repository, the Repo calls the DbContext methods - Apart from the Console App, other projects **MUST NOT** have Console.Read/Write operations - The Logic and the Console App **MUST NOT** use dbContext methods, this is only allowed for the Repository - Every layer **ONLY** communicates with the layer directly below (Upwards communication: with events - not needed now) - Usage of the entity types is **currently** allowed in all layers (this is **not** a good thing, but this semester it is accepted…) - **Be careful that exceptionally the repository should contain the MDF/LDF files as well (edit the .gitignore!)** - You must follow the SOLID principles and the basic layer separation rules! Avoid "god objects": the Repository classes must be separated according to your entities - There is no explicitly specified expectation for separating the Logic classes, use your common sense to split up the logic layer into multiple classes as well, as it fits the topic of the project work (this should not be entity-centered, according to the DDD principles the Logic classes should always be business logic centered). - It is expected that you have zero compile warnings and errors.

10/APR 23:55	Menu + All List operations	MySongShop.Data - Service-based database, filled with data from code or SQL - ADO.NET EF Core data model, with OnModelCreating() method MySongShop.Repository - We extract the CRUD operations into a separated IRepository interface - Suggested: IRepository<T>, and related entity-specific descendants - Generic implementation for the CRUD operations MySongShop.Logic - Multiple ILogic interfaces, that define the business logic operations that can be called by the console (all CRUD and non-CRUD operations that the Console App will be able to call) - Some Logic class implementations; currently all list operations should be ready, they should call Repository methods - All the other operations can be left blank - The Logic layer MUST NOT access any DbContext descendants, as a constructor parameter you must use one (or more) repositor implementations, with interface-typed parameters MySongShop.Program - Should be a simple menu-driven app - Currently only shows "This is not ready" for all not-ready menu items - All tables must have the working list functionality - All DbContext/Repository/Logic creation is done here, **MUST BE DONE** via a Factory class
24/APR 23:55	CRUD + Non-Crud functions	MySongShop.Logic - Finish all CRUD and non-CRUD operations - The results and parameters must be typed values/collections! - The queries cannot use the DbContext methods, only the data access methods of the repository - The repository must return IQueryable data that will be chained into a more complex query - Non-Crud operations MUST NOT be in the repository layer, the Logic can/should receive multiple repositories as ctor parameter! - The Logic classes MUST NOT be partitioned according to functionalities (NO CrudLogic, NonCrudLogic, CreateLogic, ReaderLogic). The Logic classes MUST NOT be partitioned according to your entities (NO ProductLogic, CarLogic). Try to think of some Business-centered partitioning! MySongShop.Program - **Obligatory menuitems/functions:** - List all entities, add/remove/modify something (**full CRUD**) - Must have minimum three data queries that use multiple tables/entities, and executes joins (and maybe groups as well) (**minimum 3 non-CRUD methods**)

08/MAY 23:55	End of project work	MySongShop.Logic - All non-CRUD operation must have a normal xxxDoSomething() and an xxxDoSomethingAsync() version. The normal method must return a collection of elements (ie. List<SomeResult>), the other method must return a Task (ie. Task<List<SomeResult>>). - From the menu in the program, both versions must be accessible, and the results must be equally displayed - The "xxxDoSomethingAsync()" version of the non-crud operations should not use the async/await keywords. Simply write "return Task.Run(xxxx)" or "return Task.Factory.StartNew(xxxx)", where xxxx is the call to the regular non-crud operation. Then in the Console App you have to use .Wait() and/or .Result to process the result of the Task MySongShop.Logic.Tests - Test the CRUD operations with a mocked repository, without Asserts only with Moq.Verify() - Test the non-CRUD operations with a mocked repository, using nUnit Asserts AND Moq.Verify() - Directly calling Mock.Object.XXX() is FORBIDDEN Repository Freeze: 08/MAY 23:55 - Students must hand in the full and correct source code via Git; also via Git the Doxygen developer documentation; and via Moodle a short, 1-2 minutes long video in which the student demonstrates the usage and the operation of all menuitems of the application. - Gitstat: true is a requirement - Project defense during the following week