# 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 exact deadlines are announced by the practice teachers.

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:

- DoxyGen generated HTML/CHM/PDF documentation
- The code must be FXCop/StyleCop validated, using the **oenik.ruleset** file
- Usage of a database + Entity Framework 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 4 projects:
  Console App + Business logic + Entities + 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).

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

Additional materials:

- oenik.ruleset
- prog_tools_en.pdf
- official subject requirements document

# 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. 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 join and/or a group by is required to get the desired output. Also, there has to be a Java web endpoint, and the console app must use the web endpoint to fetch some data.

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
- We want to use a Java web endpoint to ask for "price offers": in the GET we pass the full car name and price; the Java servlet should generate randomly five price offers and it should return five different carName + buyerName + priceOffer triplets in some XML/JSON format.

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 the same features as the ones listed above (all tables should have list/add/modify/remove features, plus three extra more complex features, plust the usage of a Java Web endpoint).

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.

| **Date** | **Name** | **Must be ready with...** |
|---|---|---|
| 15/OCT | Start of project work | Bitbucket registration, VS config, SourceTree install<br>Define name and topic<br>Find out a nice VS Solution-name (pl. CarApp) |
| 02/NOV | End of iteration 1 | Create the project structure within the git repository<br>- CarShop.Data - Class Library<br>- CarShop.Repository - Class Library<br>- CarShop.Repository.Tests - Class Library<br>- CarShop.Logic - Class Library<br>- CarShop.Logic.Tests - Class Library<br>- CarShop.Program - Console App<br>- Java/Netbeans projekt külön könyvtárban<br>NuGet/Project settings<br>- All projects should have StyleCop and Code Analysis<br>- *.Test should have nUnit (v3), nUnit3TestAdapter, Moq<br>CarShop.Data<br>- Service-based database, filled with data<br>- Be careful that exceptionally the gir repository should contain the MDF/LDF files as well<br>- ADO.NET Entity data model, sql-first<br>- The creator SQL file should be the part of this project too<br>CarShop.Console<br>- Copy the connection string from the Data project's App.config file, because only this App.config will be loaded<br>- Should be a simple menu-driven app<br>- Currently only shows "This is not ready" for all menu items<br>- **Obligatory menuitems/functions:**<br>- List all entities, add/remove/modify anything (**full CRUD**)<br>- Must have data queries that use more tables/entities, and also uses joins/groups (**minimum 3 non-CRUD method**)<br>- Must have a menuitem that will query the JAVA endpoint in the future (**Java/web interaction**) |
| 23/NOV | End of iteration 2 | CarApp.Repository<br>- The CRUD operations are put into a separated IRepository interface<br>- Optional: IRepository<T>, and related entity-specific descendants<br>- Repository implementation for the CRUD operations<br>CarApp.Logic<br>- ILogic interface, that defines the list of BL |

| | | |
|---|---|---|
| | | (Business Logic) operations (all the CRUD and other non-CRUD operations that will be accessible by the Console App<br>- Create all CRUD operations<br>- The non-CRUD operations can be left empty<br>CarShop.Repository.Tests<br>- Test the CRUD operations using some fake/mock DbContext<br>CarShop.Logic.Tests<br>- Test the CRUD operations with a mocked repository<br>CarShop.Program<br>- Make the CRUD operations work<br>**Layering rules:**<br>- The Console App calls Logic operations, the logic forwards the CRUD operations to the Repository, the Repo calls the DbContext methods<br>- The Logic and the Console App **MUST NOT** use dbContext methods, this is only allowed for the Repository<br>- Every layer **ONLY** communicates with the layer directly below (Occassional upwards communication: with events - not needed now)<br>- Usage of the entity types is allowed in all layers (this is not a good thing, but this semester it is accepted…)<br>CarShop.JavaWeb<br>- The endpoint should be ready, that generates random data based on the GET parameters<br>- Accessible from the browser (suggested: postman tests) |
| 07/DEC | End of project work | CarShop.Logic<br>- Implement the non-CRUD operations<br>- The queries cannot use the DbContext methods, only the data access methods of the repository<br>- The repository will give back IQueryable data that will be chained into a more complex query<br>CarShop.Logic.Tests<br>- Test the non-CRUD operations using a mocked repository<br>CarShop.Program<br>- The non-CRUD operations should be accessible via the menu<br>- There should be a menuitem that uses the Java endpoint to get some data (this can be refactored into the Logic, if wanted). The data should be displayed as well. |

Webprogramming and advanced development techniques - requirements