|  |
| --- |
| **EASJ Notes** |
| Databases |
| Exercises |

|  |
| --- |
| By Per Laursen  01-10-2018 |

[Exercises 2](#_Toc525904886)

[DBIntro.1 2](#_Toc525904887)

[RelMod.1 3](#_Toc525904888)

[DBVS.1 5](#_Toc525904889)

[DBVS.2 6](#_Toc525904890)

[DBVS.3 7](#_Toc525904891)

[DBVS.4 8](#_Toc525904892)

[SQL.1 9](#_Toc525904893)

[SQL.2 10](#_Toc525904894)

[SQL.3 11](#_Toc525904895)

[SQL.4 12](#_Toc525904896)

[Norm.1 13](#_Toc525904897)

[Norm.2 14](#_Toc525904898)

[Norm.3 15](#_Toc525904899)

[Norm.4 16](#_Toc525904900)

[Norm.5 17](#_Toc525904901)

[DBDesign.1 18](#_Toc525904902)

[DBDesign.2 19](#_Toc525904903)

[DBDesign.3 20](#_Toc525904904)

[DBDesign.4 21](#_Toc525904905)

[DBDesign.5 22](#_Toc525904906)

[DBDesign.6 23](#_Toc525904907)

# Exercises

|  |  |
| --- | --- |
| **Exercise** | DBIntro.1 |
| **Script** | **(none)** |
| **Purpose** | Get started on thinking about the role of a database, and on some issues relating to how a database should be used. |
| **Description** | The below questions are meant to get you started on thinking about some issues relating to databases. You may not have much specific knowledge about databases yet, so just discuss the questions on a general level. There is no right or wrong answer to the questions ☺. |
| **Steps** | 1. Consider the IT systems you are in contact with during an ordinary day. **Which of them do you think are connected to a database?** Instead of just answering *“all of them”*, try to come up with specific examples. It would also be great to find examples of systems that you think are not connected to a database, and why they don’t need to be. 2. Pick one of the systems you think are connected to a database. A game could be a good example, e.g. an online multiplayer game like World of Warcraft, Fortnite or something similar. Another example could e.g. be a web shop. **What sort of data do you think the system you selected needs to store in its database?** 3. Two main types of architectures are usually used for systems invol­ving a data­base: **two-tier** and **three-tier** architectures. **What are the main diffe­ren­­ces between these two architectures?** 4. Consider again the system you selected. **Do you think that specific sys­tem is using the two-tier or three-tier architecture?** 5. [Difficult] Consider an online game like World of Warcraft, which invol­ves many different types of data.    1. **What types of data could it be beneficial to store on the Client side (in order to avoid having to transfer too much data from the Server to the Client during the game)?**    2. **What types of data must be stored on the Server side?**    3. **Where should the business logic of the game be placed**? |

|  |  |
| --- | --- |
| **Exercise** | RelMod.1 |
| **Script** | **(none)** |
| **Purpose** | Get started on defining a relational model for a specific scenario. |
| **Description** | As some of the very first steps towards building a data model for a game, we consider two conceptual entities **Weapon** and **Character**.  The game designers describe a **Weapon** in this way:   * A **Weapon** represents a “weapon model”, i.e. a specification of a general weapon type, for which several physical instances can exist. All instances of a weapon type will, however, be completely identical. * A **Weapon** has a **name**, a **type** and a **rarity**. Examples of **type** are Gun, Sword and Mace. Examples of **rarity** are Common, Rare and Epic. * A **Weapon** has an **item level**. An item level is a positive integer value. * A **Weapon** has a minimal damage and a maximal damage. Both are measured in **damage points**, which are positive integer values. * A **Weapon** has 0, 1 or more jewel sockets. * A **Weapon** is either one-handed or two-handed.   The game designers describe a **Character** in this way:   * A **Character** has a **name**, a **race** and a **class**. Examples of **race** are Human, Elf and Dwarf. Examples of **class** are Wizard, War­rior and Hunter. * A **Character** has a number of **health points**. Health points is an integer value. * A **Character** may carry weapons in his/her hands. Both hands (left and right) can carry weapons. A one-handed weapon can be carried in one hand (left or right), while two-handed wea­pons require both hands. A **Character** can thus carry 0, 1 or 2 one-handed weapons, or 0 or 1 two-handed weapons. |

|  |  |
| --- | --- |
| **Exercise** | **RelMod.1** [Continued] |
| **Steps** | 1. Create a table definition for **Weapon**. Consider:    1. What **columns** should be included?    2. What should the **domain** for each of the columns be (i.e. what are valid **values** for each column)?    3. Which column(s) will be a **primary key** for the table?    4. Can some of the columns contain ***null*** values? 2. Also create a table definition for **Character**, following the same conside­ra­tions as for **Weapon**. However, here you also need to consider how to model the relation between **Character** and **Weapon** (hint: foreign keys). 3. Consider how to ensure **referential integrity** between **Character** and **Weapon** rows. More specifically: do we need to put some constraints on some of the columns in **Character**, with regards to the values they may contain? 4. What should happen in case a **Character** row is deleted? What should happen in case a **Weapon** row is deleted? 5. Consider how to ensure that the **business logic** (i.e. the rules described above) for the game is maintained. This could e.g. be by placing further constraints on certain columns. It may also involve defining rules for values from multiple columns. 6. [Difficult] Implement your entire data model (including constraints and business logic) in C#. This will probably involve creating (at least) two classes **Character** and **Weapon**. |

|  |  |
| --- | --- |
| **Exercise** | DBVS.1 |
| **Script** | **GameDBScript01** |
| **Purpose** | Create a database and add a single table to it. |
| **Description** | The script creates a table **Weapon** in the **GameDB** database, popu­lated with 15 rows. |
| **Steps** | 1. Open Visual Studio, and open the **SQL Sever Object Explorer** window (found under the **View** menu). 2. Expand the **SQL Server** entry, then the **(localdb)…** entry, and finally the **Databases** entry. 3. Right-click on **Databases**, and choose **Add New Database**. 4. In the **Create Database** dialog, enter “GameDB” in the **Data­base Name** field, and click **OK**. 5. Check that a new database named **GameDB** has been created in the **Databases** entry. If not, try to refresh the window by clicking the **Refresh** icon in the top left corner. 6. Right-click on the **GameDB** entry, and choose **New Query**. An **SQL Query** window should now open. 7. Now open the **GameDBScript01.txt** file (found in the **Scripts** folder under the **DB** chapter), and copy the entire content of the file into the **SQL Query** window. 8. Execute the script. 9. Go back to the **GameDB** entry in the **SQL Sever Object Explo­rer** window, and expand the **Tables** entry. A new table named **Weapon** (listed as **dbo.Weapon**) should now be found in the list of tables. 10. Right-click on the **Weapon** table, and choose **View Data**. You should now see 15 rows of data. 11. Right-click on the **Weapon** table, and choose **View Designer**. You should now see the table definition in the **Designer** view. |

|  |  |
| --- | --- |
| **Exercise** | DBVS.2 |
| **Script** | **GameDBScript01** |
| **Purpose** | Run a couple of given queries against the **GameDB** database. |
| **Description** | The script creates a table **Weapon** in the **GameDB** database, popu­lated with 15 rows. |
| **Steps** | 1. Complete exercise **DBVS.1**, such that you have created a **GameDB** data­base, and have added a table **Weapon** to the database. 2. Find the **GameDB** database in the **SQL Server Object Explorer** window (under **SQL Server** | **(localdb)** | **Databases**), right-click on it, and choose **New Query**. An **SQL Query** window should open. 3. In the **SQL Query** window, run the query: SELECT \* FROM Weapon. This query should return 15 rows of data. 4. In the **SQL Query** window, run the query: SELECT \* FROM Weapon WHERE item\_level > 20. This query should return six rows of data. 5. In the **SQL Query** window, run the query: SELECT \* FROM Weapon WHERE type = 'Gun' ORDER BY name. This query should return four rows of data, ordered in alphabetical order of the name. 6. Feel free to experiment a bit with creating and executing other queries. Also take note of how the **SQL Query** window reports syntax errors to you, and how it colors keyword, strings, etc.. |

|  |  |
| --- | --- |
| **Exercise** | DBVS.3 |
| **Script** | **GameDBScript01** |
| **Purpose** | Add a new table named **Character** to the **GameDB** database, by defining it in the **Designer** window. |
| **Description** | The script creates a table **Weapon** in the **GameDB** database, popu­lated with 15 rows. |
| **Steps** | 1. Complete exercise **DBVS.1**, such that you have created a **GameDB** data­base, and have added a table **Weapon** to the database. 2. Find the **GameDB** database in the **SQL Server Object Explorer** window (under **SQL Server** | **(localdb)** | **Databases**), expand it, right-click on the **Tables** entry, and choose **Add New Table…**. A **Designer** window should open, showing a default definition of a new table. 3. In the **T-SQL** pane of the **Designer** window, change **[dbo].[Table]** to **[dbo].[Character]**. This will change the name of the new table to **Character** 4. In the **Design** pane, change the name of the column **Id** to **id** (just to follow the convention of lower-case column names). Leave the rest of the column definition as it is; this includes keeping **id** as a primary key. 5. Now add columns corresponding to the below properties (remember to use lower-case names, and to choose proper SQL types, e.g. **nvarchar** for strings, etc.):    1. **name**: is a **string**, cannot be ***null***.    2. **race**: is a **string**, cannot be ***null***.    3. **class**: is a **string**, cannot be ***null***.    4. **health\_points**: is an **int**, cannot be ***null***.    5. **weapon\_left**: is an **int**, can be ***null***.    6. **weapon\_right**: is an **int**, can be ***null***. 6. When all column definitions have been added, click on the **Update** button in the top left corner. Check that the **Preview** dialog does not report any errors. If no errors are reported, click **Update Database**. 7. Check that a new table named **Character** has indeed been added to the **GameDB** database (it should appear under the **Tables** entry under the **GameDB** database in the **SQL Server Object Explorer** window). |

|  |  |
| --- | --- |
| **Exercise** | DBVS.4 |
| **Script** | **GameDBScript01** |
| **Purpose** | Add foreign key constraints to a table, and observe the effect. |
| **Description** | The script creates a table **Weapon** in the **GameDB** database, popu­lated with 15 rows. |
| **Steps** | 1. Complete exercise **DBVS.1**, such that you have created a **GameDB** data­base, and have added a table **Weapon** to the database. 2. Complete exercise **DBVS.3**, such that you have added an additional table named **Character** to the **GameDB** data­base. 3. Open the **Character** table in the **Designer** window. 4. In the **Properties** pane (the top-right pane without a title…), right-click on **Foreign Keys**, and choose **Add New Foreign Key**. 5. Change the suggested name to **FK\_WeaponLeft**. 6. In the **T-SQL** pane, perform these changes to the generated line:    1. Change **[Column]** to **[weapon\_left]**.    2. Change **[ToTable]** to **[Weapon]**.    3. Change **[ToTableColumn]** to **[id]**. 7. If you have done the changes correctly, no errors should be reported. Now perform steps 4 to 6 again for the **weapon\_right** column, i.e. create a new foreign key named **FK\_WeaponRight**, etc.. 8. Now update the database, by clicking **Update**. Check that no errors are reported, and click **Update Database**. 9. Right-click on the **Weapon** table, and choose **View Data**. Observe that the weapons in the table have **id** values from 1 to 15. 10. Right-click on the (just created) **Character** table, and choose **View Data**. The table should not contain any data yet. 11. Enter a row into the **Character** table (simply type values into the row which is initially filled with ***null*** values). When you enter values into the columns **weapon\_left** and **weapon\_right**, choose values between 1 and 15 (i.e. valid **id** values for rows in the **Weapon** table). This should pro­ceed without any errors. 12. Now add another row into the **Character** table, but this time use at least one invalid value in the columns **weapon\_left** and **weapon\_right**. This should produce an error… which is correct! This is due to the foreign key definitions added previously. It is now impossible for a character to use a non-existing weapon. |

|  |  |
| --- | --- |
| **Exercise** | SQL.1 |
| **Script** | **GameDBScript01** |
| **Purpose** | Try out some relatively simple SQL queries on a single table. |
| **Description** | The script creates a table **Weapon** in the **GameDB** database, popu­lated with 15 rows. |
| **Steps** | 1. If you do not already have a local database named **GameDB**, you should create such a database first. 2. Run the script on the **GameDB** database. 3. Open a **SQL Query** window. Write and execute SQL queries that return the following data from the **Weapon** table:    1. Full details for all weapons.    2. Name, type and maximum damage for all weapons.    3. Full details for weapons above item level 20.    4. Full details for weapons with minimal damage below 25.    5. Full details for weapons with item level between 10 and 25.    6. Full details for maces.    7. Full details for maces and guns.    8. Name and maximum damage for all epic weapons.    9. Name and rarity for all weapons with a jewel socket.    10. Name and maximum damage for all rare weapons that are not two-handed weapons.    11. Name and type for weapons with a name starting with “S”.    12. Name and type for weapons with a name at most 10 characters long. |

|  |  |
| --- | --- |
| **Exercise** | SQL.2 |
| **Script** | **GameDBScript01** |
| **Purpose** | Try out some SQL queries involving calculation, possibly using some of the built-in aggregate functions. |
| **Description** | The script creates a table **Weapon** in the **GameDB** database, popu­lated with 15 rows. |
| **Steps** | 1. If you do not already have a local database named **GameDB**, you should create such a database first. 2. Run the script on the **GameDB** database. 3. Open a **SQL Query** window. Write and execute SQL queries that return the following data from the **Weapon** table:    1. The minimum item level for any weapon in the table. Name the returned column **min\_item\_level**.    2. The maximal number of jewel sockets for any weapon in the table. Name the returned column **max\_jewel­\_sock­ets**.    3. The number of weapons in the table that have a maxi­mal damage above 100. Name the returned column **number\_of\_high\_­damage\_­weapons**.    4. The number of weapons in the table with a rarity above Common. Name the returned column **number\_of\_­weapons\_­that\_­dont\_suck**.    5. The average item level for all swords.    6. The total number of jewel sockets on the weapons in the table.    7. The name, and “scaled-down” minimal and maximal damage for all weapons. The “scaled-down” damage is the original damage divided by 3. Call the columns for scaled-down damage for **new\_min\_damage** and **new\_max\_damage**, respectively. |

|  |  |
| --- | --- |
| **Exercise** | SQL.3 |
| **Script** | **GameDBScript01** |
| **Purpose** | Try out some SQL queries involving ordering and grouping. |
| **Description** | The script creates a table **Weapon** in the **GameDB** database, popu­lated with 15 rows. |
| **Steps** | 1. If you do not already have a local database named **GameDB**, you should create such a database first. 2. Run the script on the **GameDB** database. 3. Open a **SQL Query** window. Write and execute SQL queries that return the following data from the **Weapon** table:    1. Full details for all weapons, ordered by name.    2. Full details for all weapons, ordered by number of jewel sockets (ascending). In case of equal number of jewel sockets, order by item level (ascending) .    3. Name and type for all weapons, grouped by weapon type, then name.    4. Number of weapons for each weapon type. Name the column containing the count **no\_of\_weapons**.    5. Average item level for weapons in each category of rarity. Name the column containing the average item level **average\_item\_­level**.    6. The number of weapons in each combination of wea­pon type and rarity (e.g. Common Guns, Epic Maces, etc.).    7. The number of weapons in each combination of wea­pon type and rarity (e.g. Common Guns, Epic Maces, etc.), but only including combinations containing at least three weapons. |

|  |  |
| --- | --- |
| **Exercise** | SQL.4 |
| **Script** | **GameDBScript02** |
| **Purpose** | Try out some multi-table SQL queries. |
| **Description** | The script creates three tables in the **GameDB** database:   * **Weapon**, popu­lated with 15 rows. * **Character**, popu­lated with 6 rows. * **WeaponsOwned**, popu­lated with 16 rows. |
| **Steps** | 1. If you do not already have a local database named **GameDB**, you should create such a database first. 2. Run the script on the **GameDB** database. 3. Open a **SQL Query** window. Write and execute SQL queries that return the following data from the giventables:    1. The name and type of the weapon that Frida uses with the right hand.    2. The name of all characters that use an Epic weapon.    3. The average item level of all weapons owned by Warriors.    4. As c), but now return the average item level for each category of rarity |

|  |  |
| --- | --- |
| **Exercise** | Norm.1 |
| **Table** | **Character**   |  |  |  |  | | --- | --- | --- | --- | | **id** | **name** | **class** | **weapons\_owned** | | 1 | Jax | Warrior | 12 | | 2 | Mjolnir | Hunter | 11, 17, 3 | | 6 | Caran | Warrior | 6, 8, 11 | | 9 | Zinuura | Mage | 6, 8 |   The primary key is (**id**). |
| **Purpose** | Convert a table to first normal form (**1NF**). |
| **Description** | The given table **Character** is not on **1NF**. |
| **Steps** | 1. Why is the table **Character** not on **1NF**? 2. Redesign the table in order to bring it to **1NF**. This may involve creating additional tables. |

|  |  |
| --- | --- |
| **Exercise** | Norm.2 |
| **Table** | **WeaponsOwned**   |  |  |  |  | | --- | --- | --- | --- | | **c\_id** | **w\_id** | **min\_damage** | **max\_damage** | | 1 | 12 | 10 | 25 | | 2 | 3 | 20 | 45 | | 2 | 11 | 40 | 110 | | 2 | 17 | 18 | 35 | | 6 | 6 | 30 | 60 | | 6 | 8 | 25 | 55 | | 6 | 11 | 40 | 110 | | 9 | 6 | 30 | 60 | | 9 | 8 | 25 | 55 |   **c\_id**: Represents a **Character** id, and is thus a foreign key.  **w\_id**: Represents a **Weapon** id, and is thus a foreign key.  The primary key is (**c\_id**, **w\_id**). |
| **Purpose** | Convert a table to second normal form (**2NF**). |
| **Description** | The given table **WeaponsOwned** is not on **2NF.** |
| **Steps** | 1. Why is the table **WeaponsOwned** not on **2NF**? 2. Redesign the table in order to bring it to **2NF**. This may involve creating additional tables. |

|  |  |
| --- | --- |
| **Exercise** | Norm.3 |
| **Table** | **JewelsInSocket**   |  |  |  |  | | --- | --- | --- | --- | | **w\_id** | **j\_id** | **match\_factor** | **added\_damage** | | 3 | 22 | 1.2 | 25 | | 6 | 22 | 0.9 | 25 | | 6 | 8 | 1.4 | 20 | | 8 | 49 | 1.0 | 60 | | 8 | 16 | 1.2 | 40 | | 8 | 22 | 0.85 | 25 | | 17 | 16 | 1.15 | 40 |   **w\_id**: Represents a **Weapon** id, and is thus a foreign key.  **j\_id**: Represents a **Jewel** id, and is thus a foreign key.  The primary key is (**w\_id**, **j\_id**)  The **match\_factor** represents a factor by which the specific jewel and the specific weapon match. The **added\_damage** can then be multiplied with this factor. Example: A jewel with **added\_damage** = 25 is added to a weapon for which **match\_factor** is 1.2 for this specific jewel. The total added damage is then 25 x 1.2 = 30. |
| **Purpose** | Convert a table to second normal form (**2NF**). |
| **Description** | The given table **JewelsInSocket** is not on **2NF**. |
| **Steps** | 1. Why is the table **JewelsInSocket** not on **2NF**? 2. Redesign the table in order to bring it to **2NF**. This may involve creating additional tables. |

|  |  |
| --- | --- |
| **Exercise** | Norm.4 |
| **Table** | **Mount**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **id** | **name** | **type** | **max\_speed** | **seats** | | 1 | Whitey | Goat | 25 | 1 | | 3 | Presto | Horse | 50 | 2 | | 4 | Golden | Goat | 20 | 1 | | 5 | Allerio | Goat | 28 | 1 | | 9 | Vanguardio | Elephant | 10 | 4 | | 13 | Buckey | Horse | 35 | 2 |   The primary key is (**id**).  A **Mount** is a creature (or machine) a **Character** can use to move around in the game world. |
| **Purpose** | Convert a table to third normal form (**3NF**). |
| **Description** | The given table **Mount** is not on 3NF. |
| **Steps** | 1. Why is the table **Mount** not on 3NF? 2. Redesign the table in order to bring it to 3NF. This may involve creating additional tables. |

|  |  |
| --- | --- |
| **Exercise** | Norm.5 |
| **Table** | **Subscriber**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **id** | **name** | **country** | **phone** | **phone\_prefix** | | 1 | Alex P. | USA | 5551236576 | +1 | | 2 | Jenny V. | Denmark | 44613407 | +45 | | 3 | Peter G. | Germany | 6973548766 | +49 | | 4 | Bruno M. | USA | 5553104560 | +1 | | 5 | Maria O. | USA | 5559812348 | +1 | | 6 | Hanne H. | Denmark | 55313429 | +45 |   The primary key is (**id**).  A **Subscriber** is a person which has a subscription for being able to participate in an online game. |
| **Purpose** | Convert a table to third normal form (**3NF**). |
| **Description** | The given table **Subscriber** is not on 3NF. |
| **Steps** | 1. Why is the table **Subscriber** not on 3NF? 2. Redesign the table in order to bring it to 3NF. This may involve creating additional tables. |

|  |  |
| --- | --- |
| **Exercise** | DBDesign.1 |
| **Model** |  |
| **Purpose** | Use rules for converting a Domain Model to a set of tables. |
| **Description** | The given Domain Model should correspond to the tables **Weapon** and **Character** we have used in numerous exercises. |
| **Steps** | 1. Use the relevant step(s) for converting the given Domain Model into a set of tables. 2. Verify that the resulting tables are indeed identical to (or at least very similar to) the **Weapon** and **Character** tables we have used previously. |

|  |  |
| --- | --- |
| **Exercise** | DBDesign.2 |
| **Model** |  |
| **Purpose** | Use rules for converting a Domain Model to a set of tables. |
| **Description** | The game designers have discovered a flaw in the previous Domain Model! They discovered that they also need to have a way to repre­sent a single, physical weapon. This changes the Domain Model, as shown above.  **Weapon** now represents a single, physical instan­ce of a weapon, while **Weapon­Model** represents a general model of a weap­on, of which several physical instan­ces can exist. For now, all instances of a specific model are however still identical, but this will change later. |
| **Steps** | 1. Use the relevant step(s) for converting the given Domain Model into a set of tables. 2. Compare the resulting tables with the resulting tables from the previous exercise (**DBDesign.1**). What are the differences? How do the differences relate to the changes in the Domain Model? |

|  |  |
| --- | --- |
| **Exercise** | DBDesign.3 |
| **Model** |  |
| **Purpose** | Use rules for converting a Domain Model to a set of tables. |
| **Description** | The game designers now introduce the concept of **jewels** into the game. Jewels can be placed in jewel sockets on weapons, thereby increasing the damage dealt by the weapon. Jewels are defined as follows:   1. A number of **jewel models** exist. Each jewel model contains a name, a rarity, and a value for added damage. 2. A number of **physical jewels** exist. A physical jewel will always be an instance of a jewel model. 3. Many physical jewels of the same jewel model can exist. All physical instances of a specific jewel model are identical.   This causes the Domain Model to change, as shown above. |
| **Steps** | 1. Use the relevant step(s) for converting the given Domain Model into a set of tables. 2. How many extra tables did you need, as compared to the result from the previous exercise (**DBDesign.2**)? |

|  |  |
| --- | --- |
| **Exercise** | DBDesign.4 |
| **Model** |  |
| **Purpose** | Use rules for converting a Domain Model to a set of tables. |
| **Description** | The game designers now introduce the concept of **matching**. Matching covers the idea that a jewel model and a weapon model form a **match** (or mismatch), which results in a modification of the damage added by socketing a jewel of a given jewel model onto a weapon of a given weapon model. This modification is expressed as a **match factor**, which is a decimal number, e.g. **1.2**.  **Example**: A jewel instance of a jewel model with **added\_damage** = 25 is added to a weapon instance of a weapon model for which the match factor is **1.2** for the given jewel model. The total damage added by socketing the given jewel on the given weapon is therefore 25 x **1.2** = 30. |
| **Steps** | 1. Use the relevant step(s) for converting the given Domain Model into a set of tables. 2. How is the new table different from the existing tables (i.e. the tables resulting from solving **DBDesign.3**)? 3. [Difficult] The game designers have defined that if no match factor is defined for a weapon/jewel model pair, the match factor is 1. Could this choice be problematic with regards to implementation (hint: queries)? |

|  |  |
| --- | --- |
| **Exercise** | DBDesign.5 |
| **Model** |  |
| **Purpose** | Use rules for converting a Domain Model to a set of tables. |
| **Description** | As a final (?) change, the game designers decide that a character can own seve­ral weapons, i.e. not just the (up to) two weapons held in hands. This results in the addition of the relation **owns** between **Character** and **Weapon**. |
| **Steps** | 1. How does this addition affect the definition of the corresponding tables? 2. The model is now considered complete. We do however also need to consider potential problems concerning **referential integrity** and **busi­ness logic**.    1. What rules should be enforced in order to ensure **referential integrity**? To start with, think about what (if any) problems might be caused by deleting a row from each of the tables.    2. What rules should be enforced in order to ensure that the **busi­ness logic** is notviolated (hint: which weapons can you use)? |

|  |  |
| --- | --- |
| **Exercise** | DBDesign.6 |
| **Model** |  |
| **Purpose** | Use rules for converting a Domain Model to a set of tables. |
| **Description** | The above Domain Model is a model for an online recruitment tool. It contains three conceptual entities:   * **Person**: A person can create a profile on the website, including a name and e-mail address (a real system will probably contain additional data) * **Company**: The system will contain representations of various companies, and a person can then specify on his profile that he has worked for spe­cific companies during specific time periods. * **Competence**: The system also contains representations of various com­pe­tences, e.g. “programming”, “accounting”, etc.. A person can then specify on his profile that he has a number of specific competences.   The model also contains three relations. The Person-Competence and Person-Company relations should be straightforward. The Person-Person relation **recommends** is a bit special, however. The idea is that one person can recom­mend another person. Both persons must have profiles in the system. |
| **Steps** | 1. Work through the entire process of creating a set of tables from the given Domain Model, including considerations for referential integrity and business logic. 2. Take a closer look at the representation of the three relations. Could it be relevant to add additional attributes to the relations them­selves (how did you e.g. represent the period a specific person has worked for a spe­cific company)? 3. [Difficult] Suppose that the recommendations are no longer just general recom­menda­tions, but rather recommendations on a specific compe­tence. How would that change the Domain Model? And the tables? |