Contents

[Software Documentation 1](#_Toc390554943)

[DB structure 1](#_Toc390554944)

[EER 1](#_Toc390554945)

[Things that need to be explained: 1](#_Toc390554946)

[Considerations in creating the DB 1](#_Toc390554947)

[SQL injection defense 2](#_Toc390554948)

[Code Structure 3](#_Toc390554949)

[GUI 3](#_Toc390554950)

[DB 3](#_Toc390554951)

[core 4](#_Toc390554952)

[Importing 5](#_Toc390554953)

[packages and classes 5](#_Toc390554954)

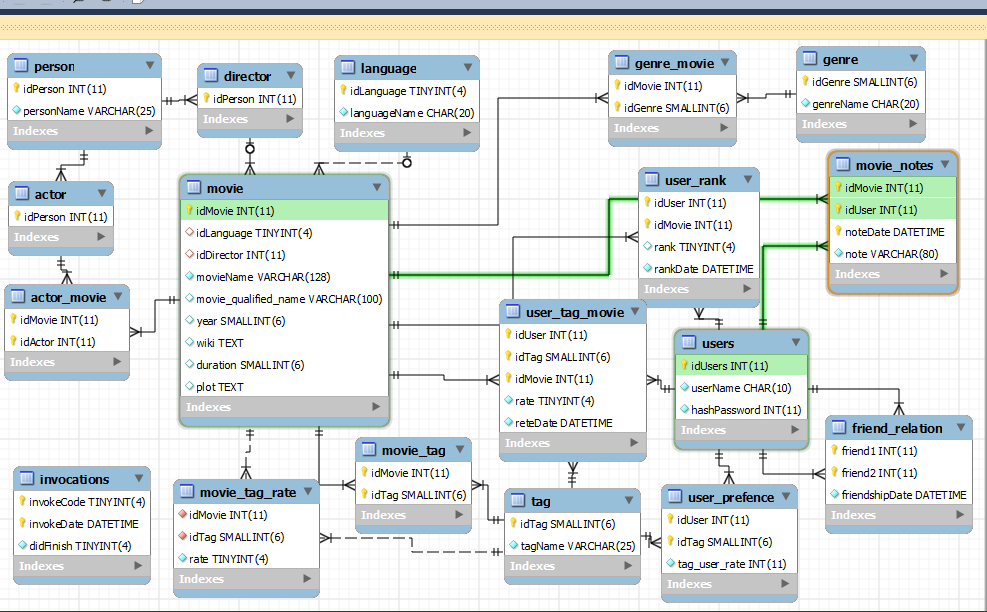
[Flow of the import process 6](#_Toc390554955)

[Miscellaneous 7](#_Toc390554956)

# Software Documentation

## DB structure

### EER

****

### Things that need to be explained:

* A tag is something that characterizes a movie (for example: 1950's, blood exc.)
* A user can rank a movie to say whether he liked it and which movie has what tag, this is how we tell witch movies he would like in the future (what movies he said he liked and what tags do they have - The preferences of users are saved in user\_prefence).
* Things that would be presented in the site as activities are saved in the data base with a date.

### Considerations in creating the DB

* **InnoDB vs MyISAM**  
  MyISAM- used on tables with no foreign keys, when more SELECTs (then INSERTs and UPDATEs) are performed on the table.  
  InnoDB- used otherwise
* **INT and CHAR sizes**we had checked what the biggest possible value is, and determined by that parameter the smallest range possible to save DB space.
* **HAR vs VARCHAR vs NVARCHAR**NVARCHAR- no Unicode was used, so no nvarchar was used.  
  CHAR - where words are not very long (<= 24) we used char to get better query speed (with low space cost)   
  VARCHAR- otherwise, to save space.
* **DATE and TIME**  
  DATETIME - the only date type used, seemingly the simplest type for usage with java
* **Redundancy issues**  
  **actors**- some persons are actors but not all of them. To get an easy way to enforce constraints their person IDs are duplicated to the 'actor' table  
  **directors**- same issue as with actors   
  **tags and genres** - some but not all of the genres are also tags but they have different duties and come from different sources

### SQL injection defense

All of our queries are executed using prepared statements, which is considered an SQL injection free mechanism

## Code Structure

### GUI

#### Classes

* Main – Consists of the global main function.
* abstract\_window – an abstract dione window, extends Shell, defines a window with Dioneicons for tasksbar
* gui\_utils – several functions used all over the gui package. Consists of the launcher of the.   
  Application, some GridData and FormDatafactories, and some more utilities.
* all\_tabs\_window, log\_in\_window, movie\_details\_window, import\_window, import\_progress\_window, preferences\_window, comment\_window, movie\_comments\_window – all these classes extends  
  abstract\_window, each consists of the relevant data.
* overview tab, search\_movie\_tab, social\_tab, recommendation\_tab, settings\_tab, are all extend Composite. Each of these classes represents a tab in the All Tabs Window, which is the main Dione window.
* Layouts - The windows and tabs are mainly use the Grid and Form layouts for it's widgets.  
  A factory for the relevant GridData and FormData is available in the gui\_utils class.
* OS Resources – as a policy, any font, color or image created in the gui classes, is disposed in the DisposalListener of the widget it is associated with. For example if a label uses font, a call to font.dispose() will take place while the label is disposing.

#### UI threading:

Every access to the DB takes place within a different thread, created just for this task. It allows greater interactivity, making the GUI responsive and fault-tolerant, when db access is required.

The java Executor framework is used, with a thread pool of 10 worker threads. The executor is initialized at the beginning, and tasks are sent to it, via asyncExec and syncExec, according to the expected length of a DB response.

In addition, the cron thread is started on init of the program, and will call functions for calculating the recommended movies every 15 minutes.

The import process is called directly via the run method, implemented by the importer class. We use the observer pattern, to update the UI of import progress, and allow for user-termination of this Thread.

#### UI-DB communication:

This communication is achieved via the BL, module which has classes that mangage all logic for movies, persons and users. For each instance of these classes, various functions for getting desired information from database are present.

## DB

This package is responsible for data base connection handling and pooling, generic statement creation and implementation of queries. basically everything related to DB.

The package assumes data already exists in tables, though it will not crash if it doesn't.

**Jdbc\_connection\_pooling**this is a singleton class that gets connections from the driver. Saves them in a vector and on a call to getConnection, returns a SQL connection.

Has two vectors one for currently available connections and the other for free ones.

**db\_operations**An abstract class inherited by all the db\_queries files, it implements the generic SQL functions (select, delete exc.) and has some cool generic functions that saves us from code duplications. Also implements the functions relevant for invocations table

**db\_queries\_**\*  
these classes contain the actual methods that query the data base. All three pretty much have the same purpose and were split because of their size.

## core

This package is used to partition between the GUI layer and the DB layer and perform whatever changes the graphic unit expects to be made on the data after it was pulled from the data base.

**\*\_logic**  
as in the db\_queries files these classes have the same purpose and have been separated from size matters.

They are the core of the business logic level, they should do the big logical actions when actually most of.

Default values (such as limits) are pulled from the config.

The user\_logic class holds the current-user-id and uses it as the default id for some functions, this is the reason it is not abstract the others are.

**core.entities**

some entities are simple, and are represented in a ID,Name map. Complex entities include movies - used mainly for parsing purposes, persons, and user-activities. light movies, are a softer version of movies, passed to the UI, that is ignorant of some parsing logic.

Entities are usually just member wrappers, with getters and setters. Movies have some "qualified name" creation logic – returning all possible name-director-year identifiers of this movie.

User activities are such as rating, tagging and commenting on movies, comply to an activity interface, and extend a relevant abstract class.

## Importing

### packages and classes

**Parsing package**:

* **Importer.java** - This class serves as a main runnable logic, that handles all of the import steps. These include Parsing YAGO and IMDB files, and loading data to tables. The importer keeps track of import progress, within each step, and notifies it's clients accordingly. The thread can be terminated in a safe fashion, terminating whatever parse or batch being done, and shutting down.
* **Iimport\_task**  - in interface for all tasks (parsers and loaders), that are used by the importer.

**Parsing.yago package**

* **Abstract\_yago\_parser.java** – an abstract class, that handles reading of a yago files (handling io, and tag-level parsing of the lines), that relies on concrete classes, that handle each specific file – react differently to tags, expect different line structures, populate different entities, etc.
* **parser\_yago\_\*** - these are the YAGO Parsers. They are responsible for our main entities – movies and persons (actors, directors), as well as relationships between these entities (who acted in a film, who directed it), and other movie details (it's creation year, it's length). Listeners can register to progress events, and termination of import, is handled at the main loop.

**Parsing.imdb package**

* **Abstract\_imdb\_parser.java** – an abstract class, that handles reading of a IMDB files (handling io and file scans). To identify a YAGO title with an IMDB title, we rely on a combination of movie name, movie director, and year. The abstract parser manages the mapping of imdb names, to yago names, taking in account the possibility of missing details, and multilingual issues. We try first to match according to all parameters, and only than allow for weaker identifications - by name and year, or name only. Listeners can register to progress events, and termination of import, is handled at the main loop.
* **imdb\_\*\_parser** - There is an extending class, responsible for parsing each file, and Concrete classes, that handle the parsing of each line: extracting facts such as languages, tags and their frequency, plots, genres, taglines and more. Two special parsers are the names and directors parsers, who's output is used to map imdb titles, to yago titles.

**Parsing.loaders package**

* **Abstract\_loader.java** – an abstract class responsible for iterating over a collection of entities (persons, movies, languages, etc…), creating a batch, executing it, and handling batch exceptions. .Listeners can register to progress events, and termination of import, is handled at the main loop.
* **\*\_loader** - Extending classes, that have some entity-specific logics – choosing whether to insert or update, creating specific prepared statements, specific batch execution, etc. there is a loader for each entity, and they can be found in the package.

### Flow of the import process

We parse Yago files, and maintain maps of entities such as movies and persons. after parsing the names and directors IMDB files, we map "imdb names" to "yago names", so we can enrich our yago titles. After the imdb parsers enrich, based on the mapping, our yago titles, we go on to load the data.

We populate tables one at a time, allowing for some flexibility and fault tolerance, if there are issues with a specific source file (say, imdb genres) or destination table. Since the importing process is synced with relevant tables, partial loads of some tables, would still allow complete loads of others, where missing values are nulled (to keep integrity). Therefore, we can easily use the auto-increment option, without constantly querying the database, so as to resolve IDs, in later loading processes. The loading process uses batched prepared statements, that are not auto-committed.

The import has several critical phases, mainly if source files are absent or empty. If critical parsers fail, the parser will terminate.

We first load directors and languages - the only foreign keys in the movie relation. we go on to load movies, and then all entities that have a many-to-many relationship with movies: actors, genres, tags.

Person entities are loaded with some care - actors and directors have an is-a relationship with persons. Thus, if a director/actor entity insert fails, we need to clean up it's person record.

If a new import is run, we maintain existing entities, based on their yago names. Movies are updated, in case that YAGO/IMDB changed their movie details. Many-to-Many relationships between movies and attributes like genres and actors, are removed, and re-established, for the same reason.

Tags are filtered from IMDB in two stages. First, we determine which tags are overall-popular: they appear in more than a predefined threshold of movies. From this pool, we choose per movie, it's top-ten Tags (based, again, on their overall frequency in IMDB) .

Tag are somewhat of an exception - they are loaded once, and assigned a default base-score. From there on, users manipulate tags and scores, and no further update will create load tags, or hinder existing ones.

## Miscellaneous

#### Data used

* YAGO data:
  1. yagoSimpleTypes.tsv - Movies, Actors, Directors
  2. yagoFacts.tsv - Actors that played in Movie, Directors of Movies
  3. yagoLiteralFacts.tsv – Lengths and Years of movies
  4. yagoMultilingualInstanceLabels.tsv – foreign names of Movies and Directors
  5. yagoWikipediaInfo.tsv - movie Wikipedia urls
* IMD data:
  1. directors.list – IMDB directors of movies
  2. genres.list – genres, and genres of movies
  3. keywords.list – tags, tag frequency, default tags
  4. language.list – languages, languages of movies
  5. plot.list – movie plots

#### External Packages

None

#### Invocations

A mechanism provided to synchronize performance of special actions between several instances of the application working with the same DB

#### Recommendations process

#### To predict which films you will like we use two mechanisms related to tagging:

#### movie\_tag\_rate (MTR) – which is the average of how well all of the users thought the tag is describing the movie

#### user\_prefence (a commemorated mistake for preference) – using the movies you rated and their MTR we try to find which tag you like the most.

#### At all times, in every instance of the application, every 15 minutes there is a check for an invocation of the filling of the table MTG (it will only happened every 15 minutes - once per DB, not per application instance)

#### In user\_prefence the rows related to each user are deleted and re-entered every time the user ranks a movie, taking to considerations the new MTR

#### General Flow of the Application

* As long as the db is empty (data import didn't happen yet), the only window that will be shown is the Import Window, which consists of only one button – "Import Data". If some user is currently making the first data import, any other user that will try to run the app will be asked to close it and try again later (in order to avoid parallel updates). After the first successful import, the next time the application will be started, it will navigate directly to the log in window. However, the user can still update data via the settings tab.
* A successful run should open the log in window. When logged in for the first time, the user is asked to rate some movies in order to help Dione to learn about his movies preferences.
* When logged in, the user's main window is the All Tabs Window.
* Any movie name shown in the overview and recommended movies tabs is clickable, and a click on it will open it's movie window.
* On a movie window, a user can open only one comment at a time, and only one recent comments window at a time.
* Settings Tab Scenarios:
* If user is clicking Log out while updating data, he will fail and will be asked to first terminate the update.
* If user is clicking the X button while updating data, the update will terminate and the application will be closed.
* Logging out from user account will immediately close all the opened windows for this user (preference window, movie details window).
* If a preference window is open, or a data update is taking place, and the user asks for it again, he will get a Failed message box.