Database Project Report

STAGE 1 - RECORDS

For the Record class I simply created a class with an array list of string field. It contains several wrapper methods to perform operations on the array list object, such as adding a data field, getting the data field at a particular index and getting the number of fields in the array list.

I decided to use an array list as opposed to a standard array because there is no restriction on how many fields it can hold and I'm not expecting there to be a huge number of field entries at this stage so that indexing speed should not be an issue.

As I'm not sure who the user of my Record class will be at this stage I have not included any error messages when accessing an out of bounds index and I am simply returning a null value if this occurs. This may need to be changed later when it becomes clearer who the client will be.

STAGE 2 - TABLE

The Table class currently holds two array list fields, one to store the column names and one to store records. Again, I decided to use array lists since they don't impose a restriction on size and will allow the creation of a table with any number of columns and records. If later it turns out that a table will contain a large number of columns or records it might be worth changing the list implementation to linked list or something else later for faster searching.

Records can only be added to a table if the number of columns matches the record's number of fields. I also decided that the column names had to be specified in the table constructor and that columns could not be added or removed at a later stage as the number of columns should always match the number of fields.

Apart from the table class I have also added two new exception classes, RepeatedColumnNameException and InsufficientColumnsInTableException. I have made these checked exceptions since they will most likely be thrown when there is some error with human user input and this should not cause the program to be terminated. These exceptions must be checked for

whenever creating a new table or calling certain methods on a table object. I have also added a MyUtilities class since there was a method in the table class that I considered generic enough to potentially be of use in other classes.

STAGE 3 - FILES

For this stage I added two new functions to the Table class to write data to and read data from an XML file. I am using the Java DOM API for this. I decided to use XML encoding as opposed to my own format because it is a standard format which allows me to side-step some of the potential problems of encoding it myself, like not being able to use control characters in fields. It is also a human readable format, so files can still be created and edited outside the program.

In addition to this, I also made some changes to the Records class. I added array notation to the constructor so that a new record can be initialized with any number of fields. I also added a lock function to the class, so that individual records can be locked and no new fields may be added. This can be used once a record has been added to a table, so that there is no mismatch between columns in a table and record fields.

STAGE 4 - PRINTING

For this stage I added a function to the Table class to print a formatted table to the terminal. Each column is formatted to be at least 30 characters wide to make sure that table elements line up. I have also added error checking for column names and record fields to ensure these are not longer than 25 characters.

STAGE 5 - KEYS

To solve the problem of a unique key for each record I initially tried to use the Java UUID class to generate a pseudo-random number that is more or less guaranteed to be globally unique. According to the Wikipedia article, [sic] Assuming uniform probability for simplicity, the probability of one duplicate would be about 50% if every person on Earth as of 2014 owned 600 million GUIDs.

However, I later realized this probably wasn't the right approach since a UUID key is much too long and impractical to use as a

reference. It also isn't necessary for a record's key to be globally unique, just unique in the table. I therefore decided to create a 4 character key (made up of a mixture of numbers and lower-case letters) that was automatically added as a record's first element when constructed. Since there is a chance of creating a duplicate key with this approach, the table object must also check that a new record's key is unique in the table before adding it.

STAGE 6 - DATABASES

For this stage I created a Database class to hold any number of tables in a HashMap collection. Functions include adding a table to the database (tables must have a unique name), removing tables and listing the number of tables. Databases can also be saved to and loaded from a folder. When a database is saved to a folder, a folder name with the .db extension is created in the working directory and all .table files are stored inside. When loading a database from a folder, an attempt is made to load all files with a .table extension into the database.

I also tried to clean up some of the code in general and implement a better strategy for handling exceptions. I created the checked exception InvalidArgumentException to be thrown by any method or constructor depending on user input directly. If there is a problem with the input, such as a field length being too long or a repeated column name, this exception is thrown and details provided in the error message. The program still continues to run though. I'm also using the IllegalArgumentException in cases where I'm not expecting input from a human user directly but some other client. This client should ensure that the input provided is correct, if not the exception should halt the program as there is clearly some logical error.

EXTENSION: STAGE 7 - TYPES

I found this stage the most difficult so far and during the course of implementation I had to add several new classes, refactor most of the old classes, make general changes and improvements to the old classes and come to a better understanding of how I actually wanted the database to work from a user perspective.

I started off by creating two main new classes, Column and Field, and the DataType enumerated type. Column and field objects store a name or a value and a supported DataType type. I also added a new method to determine the DataType type of a string, which I later placed into its own DataTypes class to keep the other classes more responsibility-driven. There is also a method in this class called accepts(), which determines if a particular data type accepts another, e.g. an alphanumeric field should accept an integer value as well as an alphanumeric value, and a method that returns a DataType type for a string representation of that type, which is mainly useful for easily retrieving types from an xml .table file.

I added support for alphanumeric, integer, email, key and foreign_key types. Foreign key types are the same as key types but with the added “fk-“ prefix. Types are determined via an appropriate regular expression and new types can now easily be added by adding the enum variable to the DataType class, adding an appropriate regular expression or other function to the getDataType() method of the DataTypes class and adding a corresponding case to the accepts() method. The new data type could then be used to set column and field types like any other.

As well as adding support for types, I had to change most of the other classes to ensure that columns and fields would only ever be able to hold values of the appropriate type. To help with this, I created a new checked exception class called UnsupportedDataType exception to be thrown whenever there is some kind of data type mismatch. I removed some exceptions like the LockedRecordException which I felt to be excessive error checking.

I also added testing for all the new classes, added test cases to the old classes and created a test suite to be able to quickly run all tests (which can also be run by compiling and running the main DB class). There is now also a demo method in the DB class to demonstrate some of the main features of the database.