DB Assignment Report

This document details the design and implementation of a home-made database system. It describes an incremental development process which broke down into the following stages:

- Records
- Tables
- Files
- Printing
- Keys
- Databases

The .java files from the first five stages of development have been included in a zip file called DevStages. Each directory name within the zip file corresponds with one of the report sections below.

The report broadly focuses on the following for each stage:

- Design decisions
- Refactoring required
- Implementation issues
- Unit testing

---

1. Records

This stage focused on developing a general-purpose ‘Record’ class which could hold any type of record as a collection of fields which are strings.

Design Decisions

The key design step was to determine how to store the fields as strings. The List interface was chosen as it allows elements to be easily inserted or accessed by their position in the list using a zero-based index.

The decision was made to use an ArrayList<String>. This was chosen over LinkedList<String> to allow for quick access of elements, given that later in the development process there will be a need to select columns in the database (and therefore a specific field in each record). However, there are obvious advantages to both.

Limited exception and error handling was introduced at this stage, given the likelihood that the methods would need refactoring later.

Unit Testing

The unit testing required at this stage was limited as the methods were very simple.

2. Tables

A general-purpose ‘Table’ class was implemented to store a collection of records as well as the field names.

Design Decisions

The key decision was how to store the column names and records. Again, the List interface was chosen as it allows elements to be easily inserted or accessed by their position.

For the column names, an ArrayList<String> was chosen (as it was for the fields in the ‘Record’ class). The records were stored in an ArrayList<Record>. The thinking was that quick access to elements of both would be needed later to, for example, insert/delete columns.

Refactoring

A method to delete fields from records was added to the ‘Record’ class after it became apparent that this had been overlooked during the first development stage. This was needed to implement a method to delete columns in the ‘Table’ class.

Implementation Issues

The main implementation issue at this stage was to ensure that the deleteColumn() method identified the correct column name and deleted the corresponding record field. This involved ensuring that mistakes were not made when determining the index used to delete fields in the records.

Unit Testing

Limited issues arose during unit testing as the methods at this stage are relatively simple. One design issue arose, which was that the methods setColumnNames() and insertRecord() could not handle single entries including a space - this would be treated as two entries.

A note was made to update this at the File development stage, as that stage would require a decision to be made on which control characters to use to separate fields.

---

3. Files

This stage focused on developing a ‘FileManager’ class to handle reading table data from files and writing table data back to files.

Design Decisions

Files hold a first line which acts as a header of column names. Every subsequent line is a record. Readable text files have been used for simplicity.

A key design decision was choosing suitable control characters. A comma was chosen to separate fields, on the basis that it is commonly used to do so (e.g. CSV files). The newline character is used to terminate records as it won’t be needed in column names or field values.

Another key decision was establishing how the data would pass between the ‘FileManager’ and ‘Table’ classes. At this point, the decision was made to create a controlling ‘DB’ class to enable this interaction.

When reading in a file, the ‘FileManager’ class stores each line as a string in a List<String>. It then returns this list to the ‘DB’ class. The ‘DB’ class then reads through the list, passing each string (or line from the file) to the ‘Table’ class for insertion.

When writing to file, the ‘Table’ class passes the table data as a string (having inserted control characters) to the ‘DB’ class. This string is then passed to the ‘File’ class, which uses the FileWriter class to write the string to file.

Refactoring

Important refactoring work was done to the ‘Table’ class at this stage to allow uploading data from files, and writing data back to files.

The insertRecord() and setColumnNames() methods were modified to use commas as the control character to separate fields in the string passed to them - previously a space was the control character.

A outputTable() method was added which returns a string containing the table data (with control characters inserted appropriately) which can then be used by the ‘FileManager’ class to write the data to a file.

Other refactoring work was done as it became clear that either existing methods in the ‘Table’ and ‘Record’ classes needed updating or that new methods were needed which had been overlooked in the first two development stages.

In the ‘Record’ class, the addField() method was updated so that it became overloaded. This enabled indexed or non-indexed insertion of fields into records.

Some error handling was also added to several methods in the ‘Record’ class, specifically those which dealt with indexed operations (e.g. deleteField(int index)). At this stage, they just printed an error message to System.err and exited the program. This decision was made because the ‘Record’ class currently interacts solely with the ‘Table’ class, and the expectation is that bounds checking will controlled by the ‘Table’ class and this keeps track of the number of columns.

In the ‘Table’ class, the insertRecord() method was updated so that it became overloaded. Again, this was to enable indexed or non-indexed insertions.

Implementation Issues

The key issue was passing the data from the ‘Table’ class to the ‘FileManager’ class. Initially, the loop which added the ‘,’ control character to records when saving a table to file was adding a ‘,’ to the end of each line. This was causing issues when the file subsequently needed to be read from again.

The fix was relatively simple, a case of removing the last comma from the string after each loop but before adding the newline record terminating control character.

Unit Testing

In the ‘Record’ class, the unit testing was updated for the addField() method to test its new overloaded functionality. The same is true in the ‘Table’ class, where the insertRecord() method was updated for the same reason.

The unit testing for the insertRecord() and setColumnNames() methods also had to be updated to account for the change in control characters at this stage in the development of the database.

---

4. Printing

A ‘Display’ class was developed to deal with printing out a table as text.

Design Decisions

The key design issues was determining how the table data would be passed from the ‘Table’ class to the ‘Display’ class.

Fortunately, the ‘Table’ class already had a method - outputTable() - which returned the table data as a string. This was developed for saving table data to a file but could also be used for printing the table data as text.

Implementation Issues

The main implementation issue was ensuring that the table columns lined up neatly.

System.out.format() enabled consistent column spacing through being able to specify a string with a minimum length, but it had one major flaw. If a string in the table exceeded this minimum length, then the columns were no longer aligned.

One possible solution was for the ‘Display’ class to scan the entire table for the longest field, and set the column widths accordingly. However, this would have required looping through the table data twice - once to set the column width and once to print the data.

A better solution was to have the ‘Table’ class keep track of the maximum field length - this is discussed in the refactoring section below. The Display class could then access this data to set the column width prior to printing out the table.

Refactoring

Refactoring of the ‘Table’ class was required to keep track of the maximum field length so that the table could be printed out in a neat fashion. A ‘private int maxColumnWidth’ field was added to do this.

The insertRecord() and setColumnNames() methods were then modified so that the length of each field added to the table was checked and maxColumnWidth updated accordingly if required.

Unit Testing

The unit testing was updated to check that the maxColumnWidth was being updated correctly when insertRecord() and setColumnNames() were called.

5. Keys

The purpose of this stage was to develop a key system to provide a more stable way of referring to records.

Design Decisions

The main design choice was how to implement a key system.

The decision was made to automatically generate integer keys when a record is inserted into the table. This removed the need for any user responsibility, and made ensuring the keys are unique relatively simple.

A private field ‘nextKey’ was used in the ‘Table’ class to keep track of what the next record’s key should be. This was incremented every time a record was added to the table. While a randomly generated key would be better for security purposes, this approach was taken at this stage to make it easier to integrate with the existing program.

A private field List<Integer> keyList was used to keep track of keys in a table and ensure there were no duplicates - particularly in files being read from.

Refactoring

Significant refactoring of the ‘Table’ class was required at this stage to implement the key system.

The main issue was reading in tables from a file, as they could or could not contain a ‘Key’ column. New methods createFromFile() and checkKey() were added to determine whether a file contained a ’Key’ column and if the keys in that column were valid. The insertRecord() and createRecord() classes were also updated as they needed to account for the fact that a record from file could or could not contain a key.

The printTable() method in the ‘Display’ class also required updating, as it became apparent that some of the formatting features didn’t take into account the number of columns in a table. This required the addition of a setNumColumns() method to the ‘Display’ class and a getNumColumns() method to the ‘Table’ class.

Implementation Issues

The main implementation issue was assessing whether tables read in from files contained keys or if they needed to be added - otherwise there was the possibility of ending up with two key columns which would have been catastrophic.

The method used to create a table from file data was updated to check if there was a key column already. If so, a checkKey() method was used to ensure the keys in the file were integer keys and it also updated the nextKey field so that records inserted subsequently would not cause key clashes.

6. Databases & User Interface

This stage involved major updates to the program. Firstly, a ‘Database’ class was added to manage a ‘Databases’ folder.

Secondly, a textual user interface was added to make the DBMS accessible.

Database Class

This ‘Database’ class can:

- List the available databases
- List the tables in a database
- Create a new database
- Delete an existing database

User Interface

The user interface was based around 3 levels of menu, which are outlined below. They were designed to allow for management of databases, tables, and the records within an individual table.

Database Menu:
1. Select Database
2. Create Database
3. Delete Database
4. Exit Program

Table Menu:
1. Select Table
2. Create Table
3. Delete Table
4. Database Menu

Option Menu:
1. Display Table
2. Insert Record
3. Delete Record
4. Update Record
5. Save Table

Design Decisions

One key design decision was how to structure the databases. To keep their management simple, a ‘Databases’ folder has been included in the ‘DB’ directory which contains all of the databases and their tables. The ‘Database’ class was designed accordingly, with its select, create and delete methods all acting solely within the confines of the ‘Databases’ folder. Each directory within the ‘Databases’ folder is a database which contains .txt files containing table data.

The other key decision was the user interface design. A system of 3 tables was decided upon, corresponding to the database, table, and record levels of the program. These are detailed above. This

choice was made to make it clear to the user which level they were working in, with the aim of preventing any confusion.

Refactoring

Significant refactoring was required at this stage to every class as implementing a user interface both led rise to the need for new methods as well as highlighting design flaws in existing classes and methods. For the sake of brevity, only a selection of this refactoring exercise is described below.

The deleteRecord() method of the ‘Table’ class was updated to take a key as a parameter rather than a row number. This was to provide a more stable way of referring to records, given that the row number can change as other records are inserted or deleted.

The setColumnNames() method of the ‘Table’ class also required changes to insert a ‘Key’ column if one did not already exist when a table was read in from file. Correspondingly, the insertRecord() method also needed updating so that it inserted a key into each record if one did not already exist.

The printTable() method of the ‘Display’ class required updating as it was discovered during testing that the number of columns in the table wasn’t being kept track of, which was ruining the formatting of the table when printed to the terminal.

A deleteTable() method was added to the ‘FileManager’ class to enable tables to be deleted from a selected database.

Implementation Issues

A major implementation issue was ensuring the user inputs did not cause exceptions once the user interface was developed. In particular, testing showed that allowing the user to name their own database folders or tables was leading to significant issues. Two approaches were taken to deal with this.

Firstly, checks were implemented in the ‘Display’ class to ensure the user didn’t input invalid database/file names etc. In particular, these checks ensured that no control characters were allowed in inputs to prevent errors with the ‘FileManager’ class at a later stage.

Secondly, when listing available databases/tables or menu options, these were presented as a numbered list. The user was then only allowed to enter an integer corresponding to the appropriate item in the list. This made the process of checking the user input significantly easier.

Another implementation issue was ensuring that the ‘Database’ and ‘FileManager’ classes were attempting to select, create, and delete databases/files in the correct directory. This process required several rounds of trial and error to get right, but was clearly important given the danger of potentially deleting the wrong file/directory by accident.

Unit Testing

Updates to the unit testing were required to take into account the changes made to methods detailed in the refactoring section above.

In particular, the unit testing in the ‘Table’ class required significant updates to take into account the fact that several methods were now using keys as their parameter rather than row numbers as they were previously.

Further unit testing also discovered several issues with ‘Record’ class methods going out of bounds which meant that these methods had to have checks implemented accordingly.

User Interface Testing

Extensive testing of the user interface was conducted which helped to uncover several bugs in various classes.

It was discovered that the deleteRecord() and updateField() methods of the ‘Table’ class were still using row numbers as their parameter rather than keys. This was easy to update.

A more difficult issue which became apparent was that the createFromFile() method in the ‘Table’ class was no longer working as intended after refactoring of other methods. This problem required a significant amount of time to resolve, as the method relied on multiple others within the ‘Table’ class. Extensive debugging uncovered that a change to the setColumnNames() method, which inserted a ‘Key’ column if one did not already exist’, had not included an update to the field numColumns, which meant that methods relying on this field were going out of bounds.