

vehicle_tax_monitoring_cw_RO KAS_RUDYS_2012818

by Rokas Rudys

FILE	25294_ROKAS_RUDYS_VEHICLE_TAX_MONITORING_CW_ROKAS_RUDYS_2012818_318857_810352363.PDF (3.08M)		
TIME SUBMITTED	15-JAN-2021 08:10PM (UTC+0000)	WORD COUNT	6280
SUBMISSION ID	142316036	CHARACTER COUNT	34586

Reflective report on implementation of interface system for road tax monitoring system

(Rokas Rudys 2012818)

10/01/2021

Words: 1490

In this assignment, I successfully created a database and java interface for a road tax monitoring system which serves the functionality of storing and handling vehicles, drivers, addresses and their tax status. This report is focused on the interface system, which uses Java and JDBC to interact with a PostgreSQL database. Overall, the project was a success and all the specified requirements, including extras, were met.

1. Java interface design

The first step of the design process is to generate required components from the requirements and include any extra features and helpers. The required features are below:

Necessary:

- Connect to the database and execute queries*
- List vehicles, drivers from the database*
- Add vehicles, drivers to the database*
- View vehicle tax close to expiry*

Extra requirements:

- Update and delete vehicles, drivers with the database*
- Toggle vehicles SORN*
- List untaxed vehicles with their drivers and addresses in a similar area*
- Generate useful statistics*

Extras for rounding product out:

- Add new tax records*
- Input validation*
- Menu, user interface printing and command inputs*

There are varying design approaches that can implement these requirements. An object-based approach would create vehicles, drivers, tax and possible menus objects. This project will use a static declarative approach, with defined modules and SQL queries.

First, we separate requirements into modules and repetition into separate modules:

- DatabaseConnection - establish and handle all connections to the database.
- DatabaseQueryHelper - perform any repetitive queries.
- Utilities - contain menus, user interfaces and input handling
- Main - provide the main menu to branch off to other selections.
- Vehicles - handle viewing, creation, deletion, updates, SORN of vehicles and assigning drivers.
- Drivers - handle viewing, creation, deletion, updates of drivers.
- Addresses - take address inputs and handle keeping them to the database
- Tax - list expiring tax, add new tax records and provide statistics.

Now we have a list of things that need to get done and where they get done, including any repetitive tasks that are combined into their modules, so all that remains is implementation. As a single developer, we created no agreed-upon data type interface, as we develop linearly, so module coupling is established as needs arise.

2. Development of search/filtering

This section will talk about the search/filtering functionality implemented in the tax class. This function fulfils the following requirements:

- **R3** It must be possible to view drivers whose road tax is close to expiry through the program.
- **R11** It should be possible to generate a list/report of drivers in a similar area with untaxed vehicles and their addresses for enforcement purposes. You may define the parameters of "similar area" but should carefully consider the intended purpose.

We assume "similar area" requirements would be best defined by the user so a search function for each part of the address allows for the most power of separation, so this function needs to do three things:

*provide a menu for listing and searching/filtering
filter already expired by similar locations
find all expiring by a certain date*

The user first accesses this function through the "list" command in the tax menu and receives a printout of all the currently expired tax, now they can enter a list of provided commands, merging these into the same section to allow searching for particulars easier.

So, in this sub menu shown below, a user can enter date to search by date, one of the address keywords or go back to return to menu, an infinite loop is used for navigation control so the user will be stuck going through various searches until back is entered to return a Boolean true that states a graceful return.

Validation is used to ensure the user cannot crash the infinite loop in some way and has to enter one of the commands, a repeated prompt is printed to the screen for all invalid inputs. Interesting quirks is that entering a newline in scanner does not trigger the loop due to the way that scanner works. 'equals("")' is required to ensure newlines are evaluated otherwise a newline will appear on the console and be added to the next input.

```

79  //
80  * Allows to list expired records, records that will expire by a certain date and expired records
81  * in similar locations by address search.
82  * @return true if exited
83  */
84  protected static boolean listTax() {
85      System.out.print(Utilities.APPLICATION_SPLIT);
86      System.out.println("Listing and searching all vehicle taxing information:");
87      String input;
88      boolean reprint = true;
89      do {
90          if(reprint) System.out.print(getAllExpiredTax());
91          reprint = false;
92          System.out.printf( "\nEnter \"Date\" to search all that will be expiring by a date\n" +
93                          "\nEnter \"Street\", \"City\", \"County\", \"Postcode\" to filter already expired vehicles by address\n" +
94                          "\nEnter \"BACK\" to leave, \"LIST\" to show all:\n");
95          input = Utility.scanner.nextLine().trim().toUpperCase();
96          //66 is "" handles entering nothing
97          if (!input.equals("")) {
98              //print all errors so can fix all at once.
99              switch (input){
100                  case "STREET" -> searchTaxAddress( column: "street");
101                  case "CITY" -> searchTaxAddress( column: "city");
102                  case "COUNTY" -> searchTaxAddress( column: "county");
103                  case "POSTCODE" -> searchTaxAddress( column: "postcode");
104                  case "DATE" -> searchDate();
105                  case "LIST" -> reprint = true;
106                  case "BACK" -> {
107                      return true;
108                  }
109              }
110              default -> System.out.println("Invalid Input! Please follow:");
111          }
112      } while (true);
113  }

```

Entering one of the keywords takes users to the two areas discussed below.

Address search/filtering

If entered an address keyword the searching/filtering for similar areas begins. This intermediary function receives an input string from the user and stays in the same category until a user enters back. The address column name is received and passed further into the query. Overall, this feature uses the visual analytics mantra “analyse first - show the important - zoom, filter and analyse further - details on demand”.

```

143  /**
144   * Takes and validates user address input and searches by its likeness
145   *
146   * FULFILLS:- R11 It should be possible to generate a list/report of drivers in a similar area with untaxed
147   *            vehicles and their addresses for enforcement purposes. You may define the parameters of
148   *            "similar area" but should carefully consider the intended purpose.
149   *
150   *            Similar area is decided by the user for their own application which is most appropriate
151   *
152   * @return true if user exited
153   */
154  private static boolean searchTaxAddress(String column) {
155      String input;
156      do {
157          System.out.printf("Enter search string for \"%s\" (Case insensitive) or \"Back\" to leave:\n", column);
158          input = Utility.scanner.nextLine().trim();
159          //66 != "" handles entering nothing
160          if (!input.equals("")) {
161              //print all errors so can fix all at once.
162              if (input.equalsIgnoreCase("anotherString: \"BACK\"")) {
163                  return true;
164              }
165              System.out.println("Following results are found:");
166              System.out.println(Utility.getTable(sqlGetExpiredTax(column, input)));
167          }
168      } while (true);
169  }

```

The below function is the JDBC call for address searching in the database, kept in the same class as this query not to be used by any other. A prepared statement protects against SQL injection attacks with a "concat" SQL command to enable the functionality of partial string searching in SQL. String combination of the column is safe because it comes from a predefined list of words that a user cannot change. Results from the database are converted to an Array List of data which will be printed as a table, the function does not print directly for reusability.

```

159  /**
160   * Search for a Tax record by column and likeness
161   * @return an ArrayList
162   */
163  static ArrayList<ArrayList<String>> sqlGetExpiredTax(String column, String like) {
164      Connection conn = null;
165      ArrayList<ArrayList<String>> results = null;
166      try {
167          conn = DatabaseConnection.getConnection();
168          PreparedStatement stmt = conn.prepareStatement("SELECT * FROM expired_tax WHERE "+column+" ILIKE CONCAT(?, '%')");
169          stmt.setString(1, like);
170          System.out.println(stmt.toString());
171          ResultSet queryResult = stmt.executeQuery();
172          results = DatabaseQueryHelper.resultsToArrayLists(queryResult);
173      } catch (SQLException throwable) {
174          System.out.println("Connection failed");
175      }
176      DatabaseConnection.closeConn(conn);
177      if (results == null) {
178          System.out.println("Failed to execute getDate query");
179      }
180      return results;
181  }

```

Overall, the above address searching/filtering provides a powerful tool to search databases for address. Splitting each column instead of searching all address columns provides some extra efficiency. “like” query is inefficient and, in the future, given the need and bigger database might need to use “startsWith” to improve slight performance. Also, returning an ArrayList of ArrayList<String> takes memory and would be cheaper to print each row, but remains for reusability sake.

Date Searching

This section of the feature achieves requirement R3. Similarly, to the one above, it takes a string input in the same way, however, before passing the parameter to the SQL query, the date is validated using a utility function that uses regex.

```

114  /**
115   * Takes and validates user date input to search for any tax records that will be expired by a date.
116   *
117   * FULFILLS: R3 It must be possible to view drivers whose road tax is close to expiry through the program.
118   * @return true if user exited
119   */
120  private static boolean searchDate() {
121      String input;
122
123      do {
124          System.out.printf("Enter date to list all that will expire by in format yyyy-mm-dd or \"Back\" to leave:\n");
125          input = Utility.scanner.nextLine().trim();
126          //&& != "" handles entering nothing
127          if (!input.equals("")) {
128              //print all errors so can fix all at once.
129              if (input.equalsIgnoreCase("Back")) {
130                  return true;
131              }
132              if (Utility.validateDate(input)) {
133                  System.out.println("Following results are found:");
134                  System.out.println(Utility.getTable(sqlGetDateTax(Date.valueOf(input))));
135              } else {
136                  System.out.println("Invalid date format!");
137              }
138          }
139      } while (true);
140  }

```

The passed date is then searched through all the data not only expired because we want to return all that may be expiring in the future or the past. Again, a similar approach handles returned results and handles exceptions.

```

121  /**
122   * Calls an SQL query to find tax records that are expired by a given date.
123   * @param date the date to search that's expired by
124   * @return the rows that qualify.
125   */
126  private static ArrayList<ArrayList<String>> sqlGetDateTax(Date date) {
127      Connection conn = null;
128      ArrayList<ArrayList<String>> results = null;
129      try {
130          conn = DatabaseConnection.getConnection();
131          PreparedStatement stmt = conn.prepareStatement(
132              "SELECT plate_number, first_name, last_name, make, model, colour," +
133              "registration_date, tax_from, tax_to, house_number, street, city," +
134              "county, postcode" +
135              "FROM detailed_tax WHERE status = 'ACTIVE' AND tax_to < ?");
136          stmt.setDate( parameterIndex: 1, date);
137          ResultSet queryResult = stmt.executeQuery();
138          results = DatabaseQueryHelper.resultsToArrayLists(queryResult);
139      } catch (SQLException throwable) {
140          System.out.println("Connection failed");
141      }
142      DatabaseConnection.closeConn(conn);
143      if (results == null) {
144          System.out.println("Failed to execute getDate query");
145      }
146      return results;
147  }

```

Overall, this feature is similar to the rest of the project implementation approach, as input validation and error handling can be seen in the entire project. It is interesting as it takes the requirements and implements them in such a way that provides more user freedom, for filtering and searching, whilst at the same time using the visual analytics mantra to help achieve more effective analysis.

3. Implementation and testing reflection

Originally the design wanted to reuse the same menu outputs and movement in the application but the number of parameters that are required, with my current understanding, to achieve this would make the code less readable, so code duplication/similarity can be seen in all classes for the menu and system outputs. Also, the same problems exist in user input as there is too much variation in the validation required to allow for easy and readable implementation with my current ability. Although in both cases, it could be possible to break the functions down reuse parts, the outside validation loop would make it less readable.

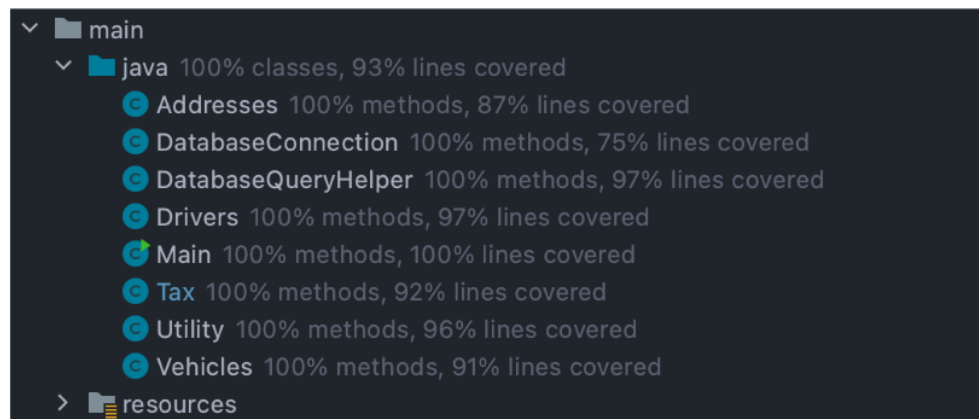
Another reoccurring problem is large methods that could be broken down such as in Drivers line 167 "getDriverDetailed()". These problems usually remain as large functions as breaking them down would make it harder to understand because it would become too fragmented and they are the only functions that need the particular validations.

Some might disagree with the use of infinite loops as it can lock a user in place so the use of instant back on failure, a count of reoccurrence or a reachable Boolean would be preferred to use. In the future, if working within a team a Boolean check approach would be safer as its more understandable, although in some use cases it was a necessity by design to never exit the loop until "back" is entered.

Code similarly to previous work is apparent, although code may be similar to previous coursework it is a matter of not reinventing the wheel, code is improved and written from scratch to perform better and as software developers its common to take previous knowledge into new projects.

Testing

Testing was performed well, using a global static Scanner user inputs could be simulated. Overall, 104 tests are created that cover 100% of methods, although class line testing is missing due to the lack of testing failed try blocks as problems were understanding and getting Mockito to work and throw expected Exceptions. Javadoc comments can be seen throughout and provide a good understanding of all the code. The test coverage execution can be seen below:



4. Final reflection

Overall, the project was a success. All essential and extra requirements were met with more improvements such as validation, error handling and added features. In addition, testing accomplished 100% method coverage with some lacking lines due to lack of exception handling knowledge. After the project, realistic, reflections were made on possible future improvements.

Overview of information hiding and software development structured design

ROKAS RUDYS 2012818

12/01/2021

words: 1407

Abstraction allows coding using programming languages without understanding hardware execution, using libraries that implement thousands of code lines without understanding what each line does and using software without understanding its programming. Overall, abstraction is the filtering of details that are not needed, it works as a multi-layered method of information hiding and is a primary factor in the progress and success of computer science [1]. To perform public abstraction, structured design and modularisation of software development is a necessity [2]. The modulation process creates general-purpose systems that work like black boxes, which is a form of abstraction, allowing developers to use systems with inputs and without knowing the programming receive the desired outputs [3]. The case for modularisation is that it improves maintainability [2], which can be the highest cost of software [4], and makes it easier to collaborate, build and replace software subsystems. This report will first provide a brief overview of structured design and move on to discuss information hiding.

1. Structured design

Implementation, maintenance and modification costs generally are minimised when each module of a system corresponds to one small problem and only has a relationship with other modules only when they independently solve a problem together [2]. Following this approach partitioning is the key to implementing a well-structured system, as it enables development and testing of modules interdependent from each other with the additional ability to test the combination of modules and the overall implementation itself [5]. Modularisation can reduce the complexity and cost of implementing software by allowing different teams to work independently, modules to become more reusable and allows module interchangeability [2]. Further, in this section, we will look at two interrelated metrics to measure code modularity.

High cohesion:

The degree of functional relatedness within a module's elements is defined as its cohesion [2]. A highly cohesive system will have modules that are focused on performing a single task which provides the benefit of higher code readability, more module reusability and easier system maintainability [3] [6]. Low cohesion systems have systems grouped arbitrarily, created large monolithic classes that are difficult to maintain, understand and reuse [6]. High cohesion creates modules that contain everything that is required in the same place, but if the code is also tightly coupled it will create a "god class" anti-pattern, where a module does too much, and the entire application is reliant on it [7]. God classes lead to higher cost as small changes can affect another part of the system unintendedly. Overall cohesion as a metric is not enough to determine software quality and reliability.

Loose coupling:

The amount of interdependence of one module with another is defined as its coupling [2]. Software coupling has three parts [6]:

- The degree, which is the number of connections between a module and others. The smaller the degree the more loose the coupling is as it allows communication with fewer interface parameters.
- Ease of connection to other modules, so the more basic the interfaces are the looser the coupling.
- Flexibility, the less dependencies for each module the easier it is to interchange.

Loose coupling interconnects components to the minimum required, where one module interacts to another through the use of an interface that does not need to be concerned with the modules innerworkings [6]. Application programming interfaces (API) create a layer of abstraction [1], allowing for separate modules to communicate with each other, permitting interchangeability of systems [3], for multiple teams to work in parallel and maintainability to improve. On the other hand, tight coupling is a type of bad programming smell known as “The Couplers” [3], it leads to unintended-side effects when making changes to the system [6], programming might become more complex as inter-dependence required other modules to be understood, more difficult to reuse and maintain [6]. However, decoupling too much can become destructive reducing the focus and coherence of each module by making classes include functions and repetitive code.

Software developers have the goal of a high cohesive and loose coupling systems, as it provides the benefits discussed and improves code maintainability [2]. Implementing these qualities can have an overhead cost associated and often is a balancing act.

2. Information hiding

Object-oriented programming provides a design pattern, that encourages high cohesion and loose coupling, where objects decouple from each other as much as possible promoting reusability and interchangeability [1]. This section will define and examine information hiding and data hiding and its use in object-oriented programming.

Data hiding and information hiding are often used interchangeably within various sources, Techopedia defines data hiding, information hiding and encapsulation as all meaning the same thing [8] whilst other sources separate data hiding and encapsulation as separate entities, such as in chapter 3 of “Java enterprise in a nutshell” [9]. Many academic papers mix the paradigms of modularity and information hiding, shown by Sawinder et al [7] abstraction and encapsulation which often are referenced as information hiding or data hiding [7]. For this report Information hiding will be defined as the principle and idea, mentioned by Colburn et al [1] whilst data hiding will be the creation of boundaries and accessors through encapsulation, similarly merged in Flangan’s “Java in a nutshell” [9] and Grady Boochs “Object-oriented analysis and design with applications” [10].

In object-oriented programming classes are a collection of data and methods that belong together to function as a single object [9], encouraging cohesion. Data hiding, also known as data encapsulation, is used to protect the internal object’s data integrity by preventing direct access [8]. In encapsulation, classes provide public methods as getters and setters that others can access, while hidden data is set as private or protected making it impossible for data to be accessed [9]. This creates a ‘capsule’ for a class that can only be accessed by provided methods in the class while the rest of the class is protected from external interference [9], which reduced coupling. The gained benefit of encapsulation is that it prevents other classes from accessing a class needlessly, allowing internal methods to be changed as long as the same data is taken and outputted but what is done with the data can be hidden [9]. Another benefit being is that the methods within the class can be reliant on private data that cannot be changed externally, preventing side effects from other classes ensuring that interdependent fields are at a consistent state [9]. Although encapsulation by itself does not provide interchangeability as any replacement of classes will require the same getters and setters with the same type of output and input data, in the same format, it may also have reliant coupling to other classes in the system that might make it difficult to replace.

Information hiding is a key component of abstraction in computer science, it is a pattern in software development that encourages reusability and changeability [1]. Information hiding attempts to create a high cohesive and loosely coupled system [6] as discussed in structured design. David Parnas who introduced the concept of information hiding stressed hiding “difficult design decisions or design decisions likely to change” [11] [12] which means more than data hiding and encapsulation [11], instead, it is the design pattern and the way that software is developed. The addition that information hiding includes is that the difference between stored and derived data, implementation details and class structure should not be revealed [11]. Information hiding should provide looser coupling and higher cohesion, in addition, output getters and setters should provide a generic type and process the data as needed within hidden methods [11]. Object-oriented programming would accomplish information hiding through interfaces, inheritance, abstract classes and polymorphism that allow classes to be replaced by communicating with just the interfaces themselves so that other parts of the application are only concerned by the output that is given.

3. Conclusion

Computer science is built on the concept of abstraction [1], and the use of structured design principles and information hiding will encourage reusability [9] and reduce costs of software development [4]. High cohesion and loose coupling are something to strive for when developing software as it improves the longevity and maintenance of code [6]. The use of information hiding will help create systems that meet those metrics but data hiding on its own is not enough. There are varying definitions of information hiding and data hiding that [3] academic sources need to come to an agreement to, so the understanding of the principles is clearer and more concise.

References

- [1] T. S. G. Colburn, "Abstraction in Computer Science," *Minds & Machines*, vol. 17, pp. 169-184, 2007.
- [2] E. Y. a. L. L. Constantine., *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Yourdon, Incorporated., 1975.
- [3] J. E. L. Stephen Clyde, "Unifying Definitions for Modularity, Abstraction, and Encapsulation as a Step Toward Foundational Multi-Paradigm Software Engineering Principles," *Conference: ICSEA 2017 : The Twelfth International Conference on Software Engineering Advance*, vol. 12, pp. 105-113, 2017.
- [4] C. Björklund, "App Maintenance Cost Can Be Three Times Higher than Development Cost," *ECONNECTIVITY*, 15 April 2019. [Online]. Available: <https://www.econnectivity.se/app-maintenance-cost/#:~:text=A%20report%20by%20Forrester%20Research,cost%20is%20due%20to%20maintenance..> [Accessed 02 01 2021].
- [5] S. Delamore, "The 5 Essential Elements of Modular Software Design," *genui*, 22 01 2020. [Online]. Available: <https://www.genui.com/resources/5-essential-elements-of-modular-software-design>. [Accessed 04 01 2021].
- [6] M. Phan, "Coupling and cohesion in OOP," 23 March 2019. [Online]. Available: <https://ducmanhphan.github.io/2019-03-23-Coupling-and-Cohension-in-OOP/>. [Accessed 06 01 2021].
- [7] S. K. Satwinder Singh, "A systematic literature review: Refactoring for disclosing code smells in object oriented software,," *Ain Shams Engineering Journal*, , vol. 9, no. 4, pp. 2129-2151, 2018.
- [8] techopedia, "Data Hiding," [Online]. Available: <https://www.techopedia.com/definition/14738/data-hiding>. [Accessed 11 01 2021].
- [9] D. Flanagan, "3.5. Data Hiding and Encapsulation," in *Java enterprise in a nutshell : a desktop quick reference*, Beijing, O'Reilly, 1999.
- [10] R. M. M. E. B. Y. J. C. a. K. H. Grady Booch, *Object-oriented analysis and design with applications*, third edition (Third. ed.), Addison-Wesley Professional, 2007.
- [11] W. P. Rogers, "Encapsulation is not information hiding," *InfoWorld*, 18 May 2001. [Online]. Available: <https://www.infoworld.com/article/2075271/encapsulation-is-not-information-hiding.html>. [Accessed 10 01 2021].
- [12] D. L. Parnas, "On the criteria to be used in decomposing systems into modules,," *Communications of the ACM*, vol. 15, no. 12, p. 1053–1058, 1972.

**DATABASE REPORTS
BELOW**

Reflective report on implementation of database system for road tax monitoring system

(Rokas Rudys 2012818)

02/01/2021

Words: 1465

In this assignment, I successfully created a database and java interface for a road tax monitoring system that has a goal of storing and handling vehicles, drivers, addresses and their tax status. This report is focused on the database system, which uses PostgreSQL for and interacts with JDBC java queries to execute changes and get data. PostgreSQL was chosen to allow functionality with macOS. Overall, the project was a success and all the specified requirements, including extras, were met. The below sections will describe the process of design, implementation, the most important improvements to query performance and provide a final overview.

1. Database Design

The design first started by selecting all the required entities for the tables, which were done through scanning the requirements for nouns, the entities found:

Vehicle, driver, road tax, address, sorn, untaxed vehicles, statistics.

We then look at the relations stated by the requirements:

Drivers can own multiple vehicles

Multiple drivers can live in multiple addresses

vehicles can be declared SORN and back into use

Vehicles can be taxed

Vehicles should have held addresses.

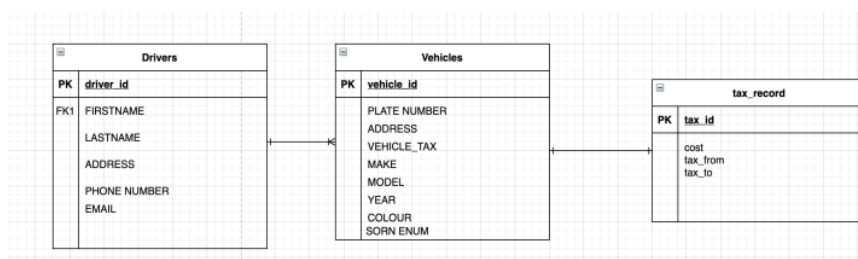
We then remove unnecessary entities for the relations so:

Statistics is removed as its analysed information about the other data.

Untaxed vehicles are those that do not have a tax, or their tax is out of date

Sorn is a state of a vehicle, so can exist as ENUM

Now we are left with tables and relations, visualising them with required data fields:



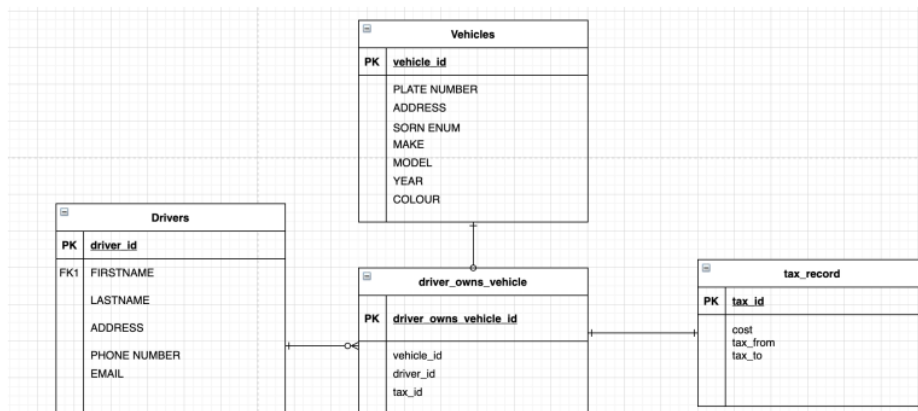
For the next step, we convert required relationships to tables:

A driver can have many cars will become its own driver_owns_vehicle table.

This also leave a design decision: should a vehicle have tax or driver_owns_vehicle.

I decided on driver_owns_vehicle as its possible for driver to change but the tax record was paid by someone else, so the new owner would tax, therefore now each owner will contain a tax record for their ownership, although this means that its possible for tax to be null when an owner has not taxed.

This has created the following changes (top of next page)



Moving onwards, we can begin normalising some data:

Address becomes its own table, at the same time it separated out into multiple fields to make possible filtering by similar location easier

We also face another design decision, should address be linked to vehicle or driver_owner_vehicle for the vehicle holding address:

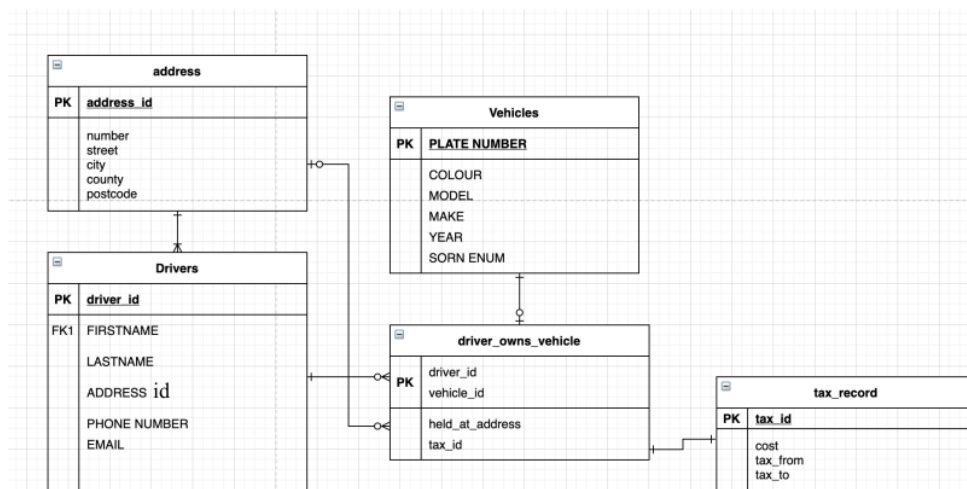
Taking the same principle as last time the holding address for a vehicle will change with the owner, so that will be linked to driver_owns_vehicle. Also, the address that a car is kept can be different from where the owner lives.

We also decide to change our primary keys to better match the data:

plate_number always unique to the vehicle so it becomes the new primary key

driver_owns_vehicle (driver_id and vehicle_id) becomes composite primary key

We could make number,street,city,county,postcode become a composite primary key but that makes using it as a key difficult.



We now can normalise this further but due to database use we won't go further:

- Street, city, county, postcode into tables that build on-top of each other
- Separate vehicle models make and year out.
- Cost into categories.

SORN could also be its own table, but no requirement to keep a log of sorn declarations and it won't have any extra fields so decided on using an ENUM.

Next step is to think about the data types to use and constraints to add to the database, going over each table, also rearranged in the order that they should be created:

Addresses:

- Address_id primary key
- Street number 20 varchar to allow for house numbers that are names
- Street, city and county 256 varchar.
- Postcode 8 varchar (maximum uk postcode)
- Postcode validation to ensure format.
- Uniqueness validations prevent duplicate addresses.

Drivers:

- driver_id primary key
- Living address Foreign key
- Firstname, lastname 256 varchar for large names
- Date of birth as date-type
- Email 320 varchar(maximum email string)
- Contact_number 11 varchar (not numeric for leading zeroes, Max UK number)
- Email format constraint
- Contact_number format constraint

Vehicles:

- Plate_number 20 varchar (max length UK) primary key
- Make, model, colour 256 varchar
- Registration as date-type
- Vehicle_status ENUM containing (ACTIVE, SORN)
- Registration constraint date not in future

Tax record:

- tax_id primary key
- cost a numeric type for maths
- taxfrom, taxto date-type
- cost_positive constraint so not negative
- taxto_constraint not later than taxfrom

Driver owns vehicle:

- Platenumber, driver_id as primary keys both foreign keys not null
- Holding address_id can be null foreign key
- Tax id can be null foreign key
- Constraint tax and plate number to both be unique independently

R9 It should be possible to update and delete drivers and vehicles.

Update drivers (Drivers class Line 335)

```
UPDATE drivers SET living_address = ?, first_name = ?, last_name = ?, dob = ?, email = ?, contact_number = ? WHERE driver_id = ?);
```

Delete drivers (Drivers class Line 365)

```
try {
    PreparedStatement prepStmt = conn.prepareStatement( sql: "DELETE FROM drivers WHERE driver_id = ?");
    prepStmt.setInt(1, driver_id);
    prepStmt.executeUpdate();
}
```

Update vehicles (Vehicles class Line 474)

```
statement( sql: "UPDATE vehicles SET plate_number = ?, make = ?, model = ?, registration_date = ?, colour = ? WHERE plate_number = ?");
    prepStmt.setString(1, plate_number);
```

Delete vehicles (Vehicles class 504)

```
PreparedStatement prepStmt = conn.prepareStatement( sql: "DELETE FROM vehicles WHERE plate_number = ?");
    prepStmt.setString(1, plate_number);
```

R10 It should be possible for vehicles to be declared SORN, then later brought back into use.

Toggle sorn vehicles depending o (Vehicles class Line 296)

```
ment prepStat = conn.prepareStatement( sql: "UPDATE vehicles SET status = ?::vehicle_status WHERE plate_number = ?");
```

R11 It should be possible to generate a list/report of drivers in a similar area with untaxed vehicles and their addresses for enforcement purposes. You may define the parameters of “similar area” but should carefully consider the intended purpose.

Allow user to search address by different selections of similar area though input:

User chooses what part of address to search and then search by what type of input they provide, which achieves an appropriate “similar area” dependant on needs.

Tax class line 346

```
( sql: "SELECT * FROM expired_tax WHERE "+column+" ILIKE CONCAT(?, '%')")
```

This will provide a list of expired vehicles in their search area.

Columns are hard coded so prep statement is not in danger of SQL injection

R12 It should be possible to generate useful statistics.

Tax class Line 277

```
public static String getDashBoard() {
    int totalVehicles = DatabaseQueryHelper.getCount( table: "vehicles", where: null);
    int totalSorn = DatabaseQueryHelper.getCount( table: "vehicles", where: "status = 'SORN'");
    int totalUntaxed = DatabaseQueryHelper.getCount( table: "expired_tax", where: null);
    int nextMonth = DatabaseQueryHelper.getCount( table: "detailed_tax", where: "tax_to > NOW() AND tax_to < (NOW() + interval '1 month')");
    int next6Months = DatabaseQueryHelper.getCount( table: "detailed_tax", where: "tax_to > NOW() AND tax_to < (NOW() + interval '6 months')");
    int nextYear = DatabaseQueryHelper.getCount( table: "detailed_tax", where: "tax_to > NOW() AND tax_to < (NOW() + interval '1 year')");
```

Calls number of queries to get the total count of tables, total count between days and basic calculations to generate the following statistics:

```
+-----+
| Statistics about taxed vehicles |
+-----+
| There are a total of 73 vehicles |
| of which 7 (9.59%) are declared SORN |
|                                     |
| Excluding Sorn:                  |
| there are currently 21 (28.77%) untaxed vehicles. |
| In the next month, there will be 0 more untaxed. |
| In the next 6 months, there will be 3 more untaxed. |
| In the next year, there will be 7 more untaxed. |
+-----+
| Enter "LIST" to see all current expired tax records |
+-----+
```

3. System performance

The use of indexing in searches drastically improves the performance of queries within the database, for example the difference between searching without indexing the date_to is shown below:

The image shows two side-by-side screenshots of a SQL IDE comparing query execution plans for the same query: `SELECT plate_number, first_name, last_name, make, model, registration_date, tax_from, tax_to, house_number, street, city, county, postcode FROM detailed_tax WHERE status = 'ACTIVE' AND tax_to < '2020-01-01'`.

NOT INDEXED (Left): The query plan shows a sequential scan on the `vehicles` table, followed by several index scans on `driver_ownership`, `tax_record`, and `address` tables. The execution time is 3.273 ms.

INDEXED (Right): The query plan shows a hash join operation, which is significantly more efficient. The execution time is 0.383 ms.

The indexed query performs $3.781 / 2.606 = 1.45$ times faster and that's on a relatively small database. But even better improvements are found by indexing the address searching feature:

```

5
6  -- efficient btree searching using indexes
7  CREATE INDEX date_search ON tax_record((tax_to::date));
8  CREATE INDEX street_search ON address(street);
9  CREATE INDEX city_search ON address(city);
10 CREATE INDEX county_search ON address(county);
11 CREATE INDEX postcode_search ON address(postcode);

```

The benefit is gained further as only one column is searched at a time and the use of a single wildcard `SEARCH%` allows for indexing, whereas the use of `%SEARCH%` would not allow it.

Performance gains occur because the searching does not have to visit every row thanks to the created data structure and the direct pointer to the location, although this has the cost of memory space, which might be difficult in larger database. Further use of caching and more refined joins can improve performance more.

4. Overall Reflection

Overall, the project was a success and the database interacted well together with the java interface, although the database has lots of possible further improvements including normalisation. System performance has been improved thanks to indexing, also the use of views simplified query building but there are more performance gains that could be had. There are also design improvements that could be made, such as tax records to keep a vehicle primary key so it can be used as a log of taxation.

Importance of ACID transaction processing to concurrent database systems.

(Rokas Rudys 2012818)

04/01/2021

Words: 1343

1. Introduction

Concurrency is when multiple operations are run and overlapping at the same time [1]. Databases often have multiple simultaneous users and changes, creating a risk of failure and the necessity for concurrent database management systems (DBMS). DBMS's support a high-level processing language such as SQL, controls database access and provide reliability [2]. Series of actions, within a database, are called transactions that are more than a statement. Transactions encase actions as logical units of work that can only either be committed or aborted. The transactional processing system (TPS) handles concurrent and non-concurrent transactions, using a concurrency control subsystem, and provide failure recovery using the recovery subsystem [3]. This report will discuss the effectiveness of an existing TPS standard "ACID" in handling different types of transactional failures with a focus on issues created by concurrency. Database failures can occur for a range of different reasons, a DBMS needs to recover if any of these occur and ensure that the system is not affected, some of these issues are mentioned below:

1. Transaction failure through logical errors caused by some internal error or invalid statement [4].
2. Syntax error, making a database unable to execute a transaction [4].
3. Concurrency failure when multiple users have conflicting changes in the database or a deadlock [1].
4. System crashes when database hardware or software breaks during transaction execution [4].
5. Disk failure caused by inaccessible storage-media possibly due to corrupted or incomplete memory [4].

2. ACID Transaction processing

The standard "ACID" is an acronym that stands for atomicity, consistency, isolation and durability. ACID-compliance is important for database systems as it helps ensure that databases remain accurate and consistent despite any database failures [3] or concurrent use. In this section, we will go over each property its concurrency, importance and typical implementation.

- Atomicity

A Database has a read and update operation, multiple complex logical combinations of these operations are used in order to perform complex. Atomicity means these operations and actions are combined as transactions that work together as a single logical unit that can either be fully completed or not at all, if it fails in the middle it cannot be left in an inconsistent and would be rolled back [5]. In concurrency, atomicity is necessary as it allows to isolate each transaction easily. Consistency is achieved by using a read-copy-update mechanism where the initial value of the database is stored or logged before any update transactions. Transactions written in SQL are typically INSERT, UPDATE and DELETE commands but more complex transactions that rely on changing the value of multiple rows in the database would require a manual declaration of a transaction to ensure it can be rolled back easily upon failure.

- **Consistency**

Any change must not break the database, data must follow the same existing rules before and after any changes. DBMS defines and ensures data consistency through the use of constraints, cascades and triggers that all validate data in some form. Although a database cannot completely guarantee the correctness of data, it protects against application errors [2]. In addition, other ACID properties ensure that data is always at a consistent state and rolled back if consistency is not met. Concurrent databases need to put in extra work to ensure that databases are recovered back to a consistent state and that transactions are executed serially [2].

- **Isolation**

Modern databases often execute transactions concurrently, meaning that multiple transactions are reading and updating at the same time. If a transaction that is undergoing changes is read in the middle by another transaction, then one of them can overwrite each-other [4]. Isolation should be used to ensure partial effects of incomplete transactions are not visible to others and that transactions execute independently from another [2]. Concurrent transaction execution should produce the same results that sequential execution would create [4]. The concurrency subsystem has the main goal of concurrency control, which will be focused on in section 3 of this report.

- **Durability**

The database should permanently record committed transactions, and any system failure should not overwrite or remove it. The recovery system ensures this by recording transactions in non-volatile memory, meaning that if there is a system crash or power outage, the progress of a transaction is recorded and can be rolled back or completed leaving the system in a consistent state [3].

ACID transactional processing properties are reliant on each other. For example, isolation assumes that a transaction is atomic meaning that a full job is not two separate transactions where concurrency would fail. Similarly, consistency relies on durability, isolation and atomicity; the other three properties ensure that all transactions will leave the database in a consistent state. Furthermore, durability requires atomicity and consistency, so it's possible to roll back failed transactions, and isolation to prevent transactions from interfering during recovery. There is even more reliance than stated here. In summary, to have a reliable and effective database, each of the ACID properties is important to a database.

3. Concurrent access control

Executing transactions in sequential order one at a time would guarantee the reliability and effectiveness of database systems. However, many users use database systems with many overlapping transactions, making concurrency necessary to modern databases. With no concurrency control, transactions could overwrite each other losing updates, could perform a 'dirty reads' in a middle of a failed transaction and order of transactions may be executed incorrectly leaving the database in an inconsistent state [4]. Concurrent control mechanisms are used to prevent any of these issues and have two major goals:

1. **Serializability** ensure concurrent databases are equivalent to a serial schedule. Transactions remain isolated from one another and are executed in order. Serializability is extremely important in monetary or other order dependant systems [1].
2. **Recoverability** ensures executed transaction returns the database to a consistent state even if it has failed or aborted [6].

Both of these goals create ACID-compliance and rely on the ACID properties. A common access concurrency method is two-phase locks; a transaction is either read locked or write-locked by a transaction until completion, preventing other transactions from interfering until it is complete. Furthermore, multiple transactions could be collated and executed in the future as a single bulk transaction [7]. There are varying approaches to using two-phase locks although are generalised into categories:

- **Pessimistic** is where exclusive locks are acquired at the start of the transaction ensuring no other operation can be performed until the transaction aborts or is committed. It's possible that a lock is acquired by a separate transaction on a required resource causing waiting times or even deadlocks if the other transaction required the current resource, making so neither transaction can proceed [4].
- **Optimistic** where the state of the data is saved at the beginning of a transaction and compared at the end, if there is a difference, the transaction is aborted and tried again. Although databases that update frequently may perform worse, this process also risks sacrificing order [4].
- **Semi-optimistic** is where transactions sometimes acquire locks in the beginning dependant if they may cause violation of some rule but do not block in other situations using the optimistic approach. This has the benefit and the disadvantages of both [4].

4. Competing Standards and Conclusion

Many NoSQL databases forgo ACID compliance in place of BASE, which stands for basically available, soft state and eventual consistency. The base approach is less strict and provides much looser consistency than an ACID system normally would, by providing inconsistent caches but providing the promise that the database will be collated at some point and reach an eventually consistent state [5]. BASE systems are designed to scale with quantities of transactions and users, although the ACID system is much more consistent and reliability of data. It's also possible to use the ACID system in a distributed model that can handle more users but runs into other issues [6]. Overall, ACID was introduced in the 1980s and is still widely used, showing the reliability and importance of ACID is to some concurrent database systems, especially those that require immediate consistency.

References

- [1] ORACLE, "Data Concurrency and Consistency," ORACLE, [Online]. Available: https://docs.oracle.com/cd/B19306_01/server.102/b14220/consist.htm. [Accessed 01 01 2021].
- [2] C. J. Date, SQL and Relational Theory, 2nd Edition, O'Reilly Media, Inc., 2011.
- [3] T. a. R. A. Haerder, "Principles of transaction-oriented database recovery," *Association for Computing Machinery*, vol. 15, no. 4, pp. 287-317, 1983.
- [4] E. DIDIK, "WHY IS CONCURRENCY CONTROL NEEDED?," Information System, Binus University, 30 08 2018. [Online]. Available: <https://sis.binus.ac.id/2018/08/30/why-is-concurrency-control-needed/>. [Accessed 01 01 2021].
- [5] Ian, "What does ACID mean in Database Systems?," Database.guide, 20 June 2016. [Online]. Available: <https://database.guide/what-is-acid-in-databases/>. [Accessed 2021 01 01].
- [6] S. Choudhury, "6 Signs You Might be Misunderstanding ACID Transactions in Distributed Databases," yugabyteDB, 22 July 2018. [Online]. Available: <https://blog.yugabyte.com/6-signs-you-might-be-misunderstanding-acid-transactions-in-distributed-databases/>. [Accessed 2021 01 01].
- [7] A. & K. B. & L. K.-Y. Chiu, "Comparing two-phase locking and optimistic concurrency control protocols in multiprocessor real-time databases.," *Parallel and Distributed Real-Time Systems, Workshop*, vol. 0, pp. 141-148, 1997.
- [8] V. Mihalcea, "A beginner's guide to ACID and database transactions," Vlad Mihalcea, 26 May 2020. [Online]. Available: <https://vladmihalcea.com/a-beginners-guide-to-acid-and-database-transactions/>. [Accessed 03 01 2021].

FINAL GRADE

/100

GENERAL COMMENTS

Instructor

Programming and Software Testing mark: 92

Database Systems mark: 93

Implementation (50): 47

Programming and Software Testing Report (50): 45

Database Systems Report (50): 46

Excellent work, Rokas. An advanced implementation, well constructed and exploring a number of topics outside the scope of this introductory module.

The reports are well written and cohesive, a wide range of appropriate sources selected and utilised. Very well done!

As a minor suggestion for improvement, aim to work on your academic writing slightly - some of your phrasing is a tiny bit clumsy in places. This is nothing other than nitpicking, however!

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18
