Kartastic Report

by Rokas Rudys

FILE	25294_ROKAS_RUDYS_KARTASTIC_REPORT_312866_1036490932.PDF (1.32M)		
TIME SUBMITTED	11-DEC-2020 04:57PM (UTC+0000)	WORD COUNT	2489
SUBMISSION ID	139293729	CHARACTER COUNT	12525

The Design and Implementation of “Kartastic”, a vehicle rental management system.

Rokas Rudys (2012818)
Module: ACCA7000, ACCA7001, ACCA7002
Degree Programme: Software Engineering (Web Development) MSc
Lecturer: Dr Tim Bashford/ Dr Michael Dacey / Dr Beverley Holland
Team Members: Riley Croucher, Zack Costello and Ridwan Chowdhury

1. Introduction

This reflective, yet hopefully short, report will be regarding a week-long agile sprint to develop a function database that is controlled through a Java console application. The random team voted on a scenario to create a vehicle rental management system, where customers book rental vehicles across a multitude of locations and managers deal with issues. Using basic requirements, we designed the application and database from the ground up, and then implemented using agile development principles.

2. Software design evaluation

This section will discuss, the design decisions made, why they were made and how agile is involved throughout them all.

2.1 Scenario Requirements

The chosen scenario had design requirements that were interpreted and understood in a certain way. Typically, clarification in an agile development model, such as scrum, would include the product owner within the design. Not having that, caused problems with interpretation such as what is meant by “*vehicle collection*”, *“suspended”* or “*payment details*”. Not having a real business to model, no product owner and no time to waste, the team decided to agree on something. This illustrates, the real benefit of agile software models focusing on customer collaboration rather than contract negotiation, as the final product would be of a much more “expected” quality with a direct, understood product owner.

Furthermore, due to lack of understood separation in requirements, it became difficult to decide what a customer's functionalities are and what managers functionalities are, this may cause some issues with the design expectations, further on. We overcame these challenges by communicating together and voting on everyone's ideas.

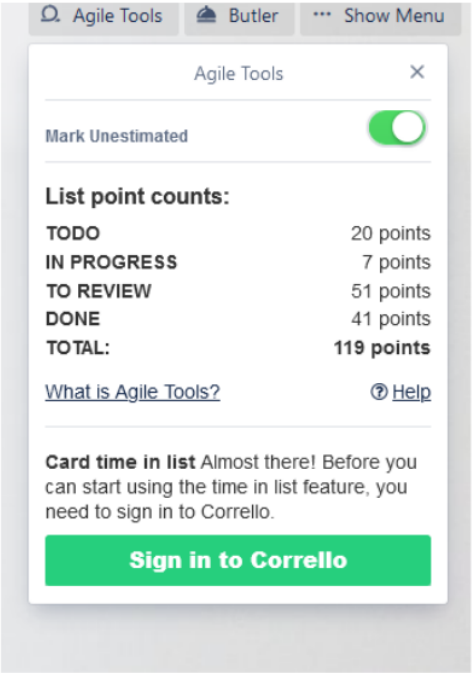
2.2 Agile process

Our team had decided to incorporate agile development in the way that we find it most useful. During our sprint planning, we decided to use User Stories, to map the features based off our understanding of the scenario requirements. We used Trello (Appendix 1) to place the stories onto a Kanban-like board, and agreed work in limit of 2, this allowed people to work at their own pace which may have caused some people to do more than others but proved more effective in getting features complete.

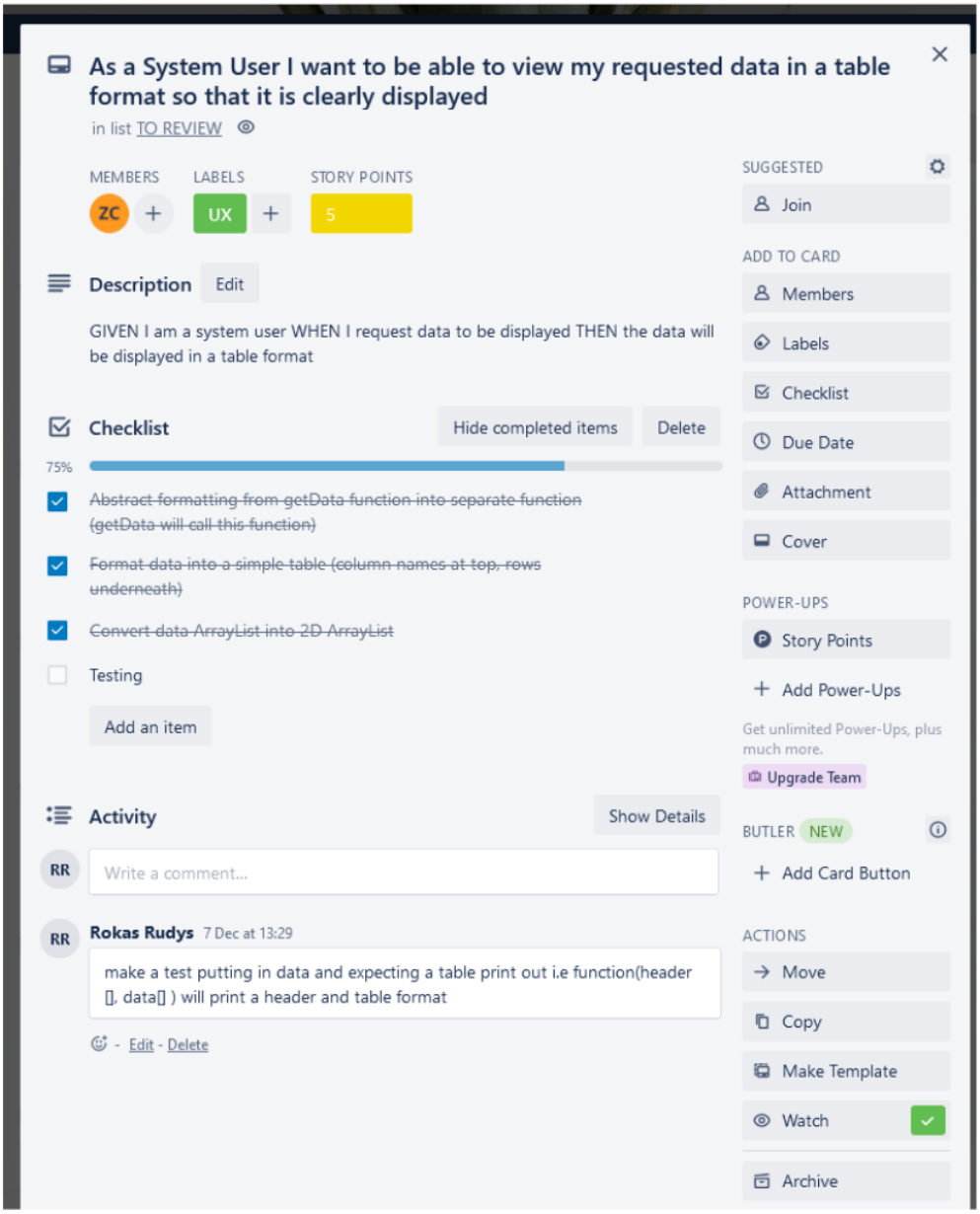
2.3 User Stories

To ensure user stories were of high quality, we defined a Definition of Ready, and for overall quality we defined a definition of done, which is available on our GitHub's readme (Appendix 2). This helped ensure that the stories that we work are implementable and that we know what to do during story review.

Each story had user story points agreed upon the whole team, using a Fibonacci sequence. The total points created was 119. As we previously have not worked on an agile project like this before we were unaware of our sprint velocity, making it difficult to estimate the user points. Considering it's a 5-day sprint, we realized stories would not get completed as we were treating the sprint backlog as a product backlog, but this was already a couple of days after starting, creating



major issues when trying to produce an overall minimal viable product.
In addition, during sprint planning we categorized each story to its believed area, using colors (Appendix 3) which helped people choose their most relevant user story next. Furthermore, during design of the system, we came to realization that there will be a lot of code repeats, so we added additional tasks to handle repetitive tasks, so where the user stories needed that function, it stayed as a dumb call to a function, allowing for iterations of work to improve the overall product. One such user story can be seen here, which prints tables from a given data call:

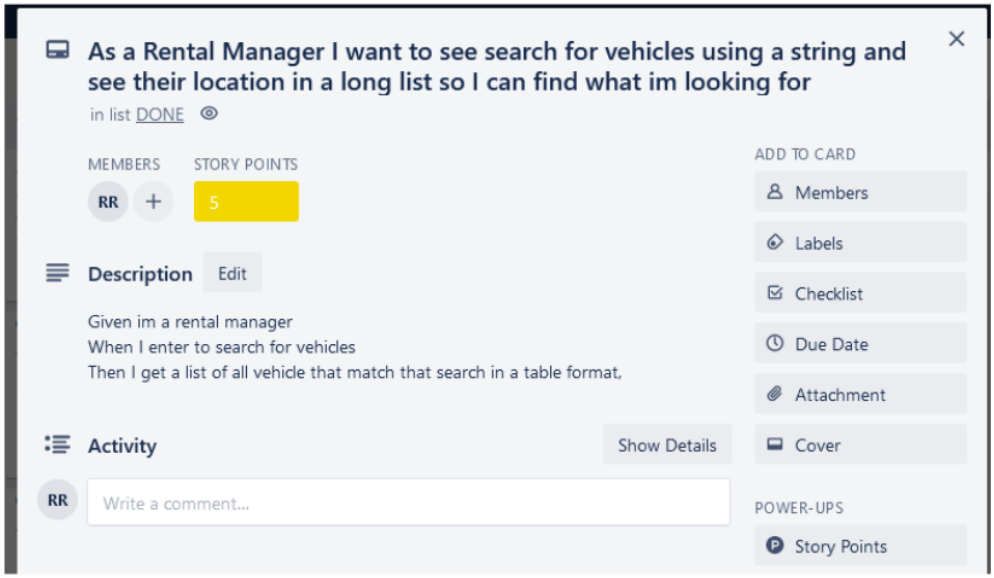


The above story shows, how Trello has helped us track our progress throughout development, and it also shows how each story was given a testing criterion as according to our definition of ready. Although discussing the code during sprint planning, typically does not happen, leaving comments helped us share our knowledge on how to implement things saving overall time in research, especially so when considering we are inexperienced developers.

2.4 User Stories

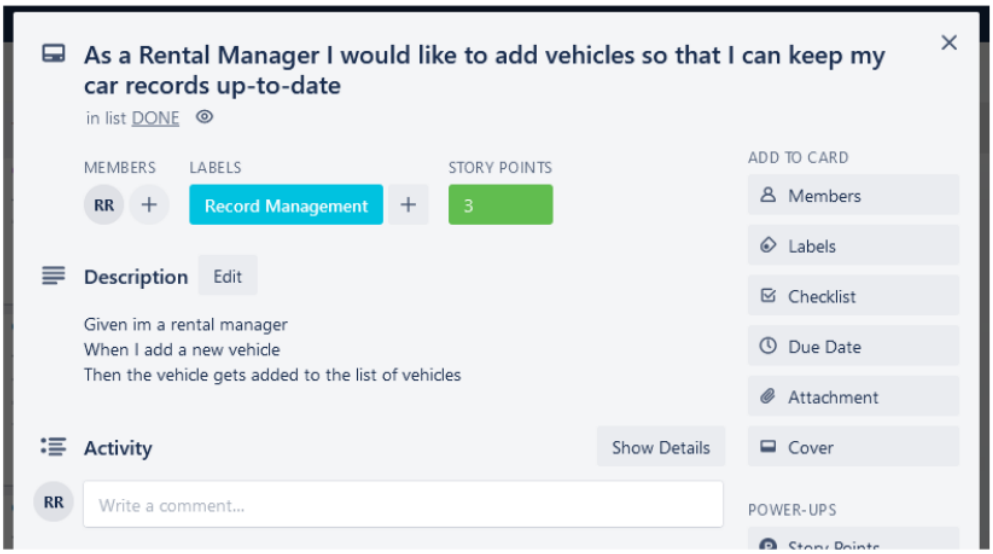
This section will overview some user stories, their design decisions, their quality and the estimates that were conducted.

2.4.1 Vehicle Search



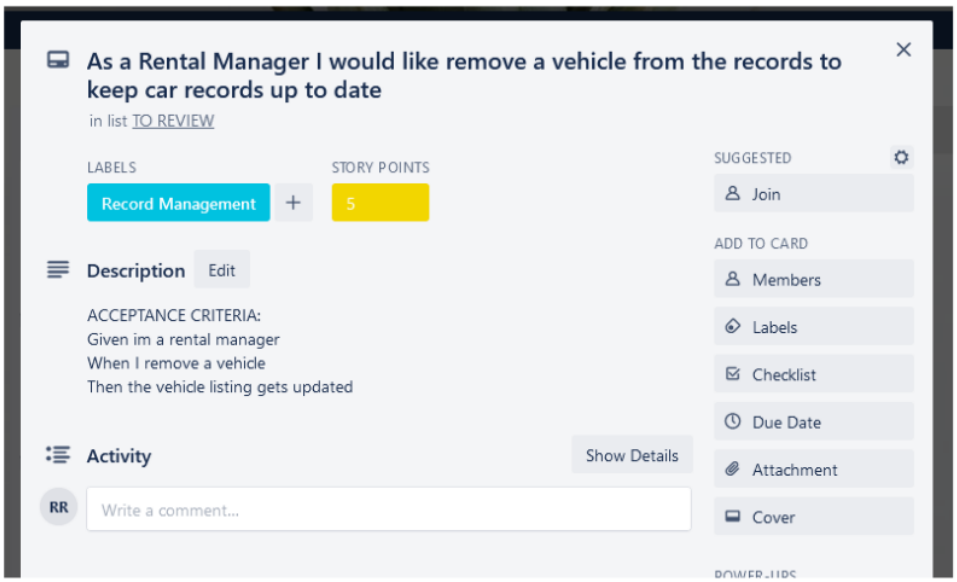
The image shows a user story for searching vehicles, this is a repeated requirement, which in future iterations should receive a templated function. This user story fulfills the requirements by allowing users and managers to find a desired vehicle to book, remove, add maintenance, swap, request vehicle collection and find availability. It's got a medium story rating, as it has to combine multiple tables to provide useful information and due to the range of inputs and validation that a String search introduces. Following the definition of ready, the story has a testable criterion, is independent, self-explanatory and understood. After implementing this, the function was adapter for other search components.

2.4.2 Add vehicles



Again, this user story should be simplified in the future through a creation of a create object template, within this project technical debt was building as the functions became more advanced and supported better validation. A function should be reworked to allow for a creation call and certain input variables that could be repeatedly used throughout the program. The story exists to fulfill the requirement *"the ability to add a new vehicle"* and meets the definition of ready and definition of done. Its currently rated at 3 story points, but this is an original underestimate considering the difficulty faced for this story it should have been more of a 5 or even an 8. Although underestimating is to be expected with inexperienced estimators.

2.4.3 Remove Vehicle



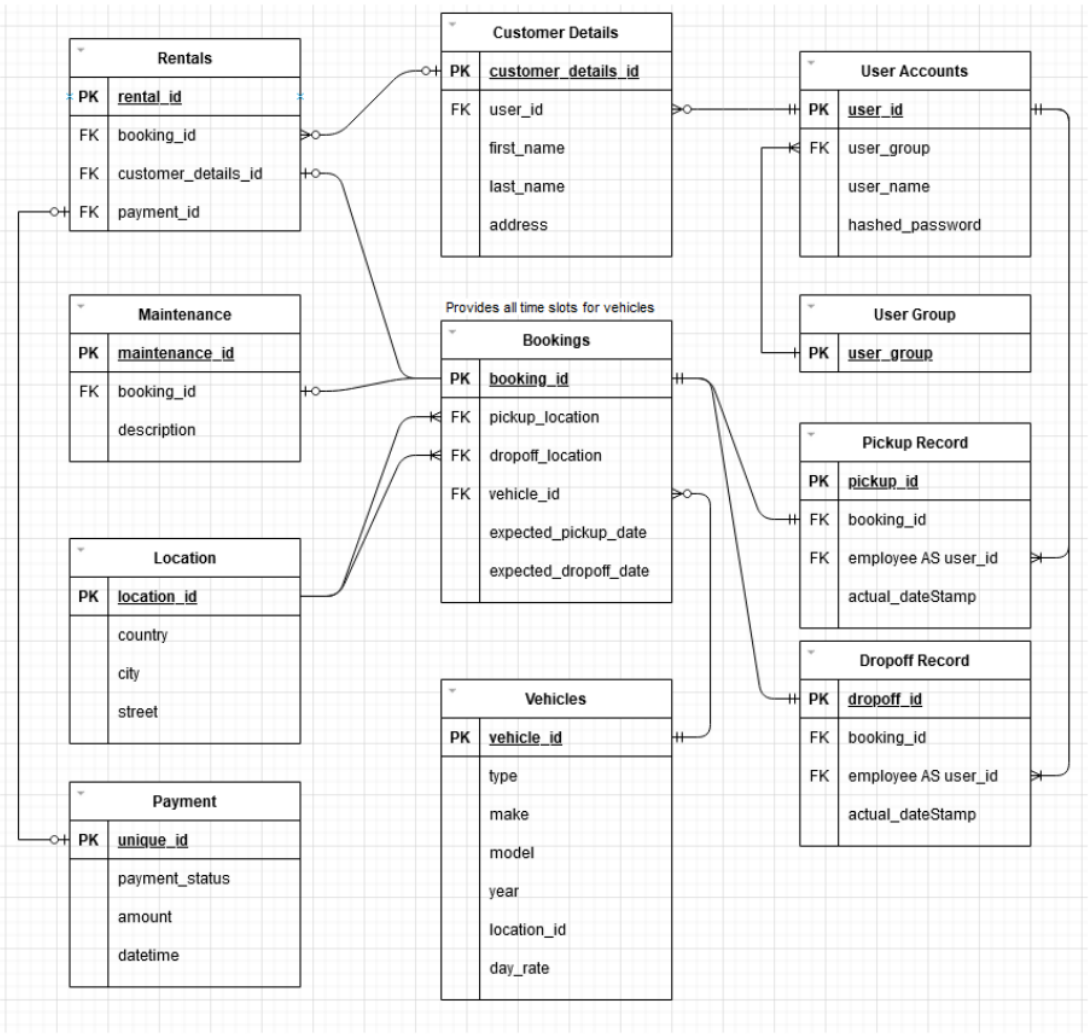
This story is similar to the above two, but instead of getting data, it is performing a delete operation. Although it is in its second iteration and does not handle the aftereffects of removing a record, through our design decisions a suspend operation would be sent before removal to inform a user that their rental is suspended, once a certain time passes it would only then get deleted or even then stored in a log if it was a professional use. Overall, the story has a high story point count as delete operations were riskier in the team's thoughts in the beginning of the Sprint, but now the input validation that it has is worth 5 story points in my opinion but not the deletion itself.

2.4.4 Overall Agile Evaluation

The team, including me, have all become good users of version controlling (GIT), which has reduced a lot of the pain of continuously integrating projects, by having a pull request restriction set up on GitHub to ensure code is checked before being merged, higher quality code is typically produced. Although there are still struggled with my personal use of agile development and my lack of iteration over smaller increments, instead my iterations tend to be larger and more polished which sometimes leads me to wasting time, although it is slowly improving. When considering this sprint, there were difficulties in understand project requirements and lack of separation of product backlog and sprint backlog which has caused too many stories to be taken on at the same time, that's why the sprint was not fully complete, and tasks are left over, in addition with my own desire to polish work unnecessarily.

3.0 Database evaluation and implementation

At the beginning of the sprint planning, we realized the complexity and size of the database portion of this project, after sprint planning, we produced a relational database overview that was normalized to third normal form. This Database diagram (Entity Relationship) visualizes the relationships between tables using standardized notation arrows, as well as lists table articles. This showed a transparent overview of the database and allowed everyone to understand how it is worked Further in this section we will discuss some of the design decisions made for the tables.



3.1 Table decisions

It's difficult to decide how table relations are structured without having the actual business data usage. We began designing the database by listing the nouns that are entities, and descriptive words that are fields, and then discussing the relationships between them. After drawing the nouns and linking them together, we discussed what fields each entity will require. After all entities were filled with fields, we then removed transitive properties where data could be acquired through a join and added foreign keys. Finally, we discussed if fields need any form of validation, such as not being null.

3.2 Database Implementation

We used PostgreSQL as our database because members of the team have previous experience working with it and its compatible between multiple platforms. We began using Databases through java JDBC, this is driver that communicates with an SQL server and can send it SQL commands and receive back Result Sets. As this was the only way we knew how we got too far in to change it to any other way, which made things go through a helper class and java seeding classes.

3.3 User Story Queries

This section will overview the previous user stories and overlook their database implementation

3.3.1 Vehicle Search

3.3.1.1 Table creation

```
stmt = c.createStatement();
String sql = "CREATE TABLE vehicles " +
        "(" +
        "vehicle_id     INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY," +
        "type           VARCHAR(50) NOT NULL, " +
        "make           VARCHAR(50) NOT NULL, " +
        "model          VARCHAR(50) NOT NULL, " +
        "year           INT NOT NULL, " +
        "location_id    INT NOT NULL, " +
        "day_rate       decimal NOT NULL," +
        "CONSTRAINT fk_location" +
        "   FOREIGN KEY(location_id)" +
        "       REFERENCES location(location_id))";

stmt.executeUpdate(sql);
```

Vehicle Table is created in the DatabaseSeeder.java, which calls a JDBC create query to make the table, which can be seen above. The table creation follows the ER diagrams and provides some validation. Table creation is tested by a JUnit test that tests all table creations at the same time, as can be seen below:

```
/**
 * Below Tests cover 100% of database seeding, excluding examples and exception handlers
 */
@Test
void testCreatingTablesIfTablesExistShouldFail() {
    DatabaseSeeder.createTables();
    assertFalse(DatabaseSeeder.createTables(), message: "Table creating with existing tables should fail");
}

@Test
void testCreatingTablesWithEmptyDatabaseShouldPass() {
    assertTrue(DatabaseSeeder.createTables(), message: "Table creation with empty database should succeed");
}
```

3.3.1.2 Table seeding

```
Faker faker = new Faker();
String[] vehicleTypes = {"SEDAN", "COUPE", "HATCHBACK", "COUPE", "SUV", "MINIVAN"};
String[] vehicleMake = {"HONDA", "TOYOTA", "BMW", "MERCEDES", "FORD", "VOLKSWAGEN", "SKODA", "MINI"};
String[] vehicleModel = {"A-CLASS", "B-CLASS", "C-CLASS", "D-CLASS", "F-CLASS", "G-CLASS", "E-CLASS", "X-CLASS"};

for (int i = 0; i < NUMBER_OF_VEHICLES; i++) {
    String sql = "INSERT INTO vehicles (type, make, model, year, location_id, day_rate) VALUES" +
            "('" + vehicleTypes[faker.random().nextInt(vehicleTypes.length)] + "'," +
            " '" + vehicleMake[faker.random().nextInt(vehicleMake.length)] + "'," +
            " '" + vehicleModel[faker.random().nextInt(vehicleModel.length)] + "'," +
            " " + faker.number().numberBetween(2000, 2020) + "," +
            " (SELECT location_id FROM location ORDER BY RANDOM() LIMIT 1),"
            + faker.number().numberBetween(20, 2000) + ")";
    stmt.executeUpdate(sql);
}
```

Table seeding is used to fill tables with fake information. The above code inserts a given number of rows into the table with each new row being completely random and different from the other, this creates realistic fake data that varies every test, ensuring robustness. The benefit of using a programmatic approach like this instead of a Query table is that it can be accessed from within the program easily, and modified by user inputs, although for development this was not necessary. It is also Tested by Junit to ensure that it runs into no problems and has error handlers during execution.

3.3.1.3 Search function

```
/**
 * Given a search time see if anything within the vehicles database matches
 *
 * @param searchTerm - the string to search
 * @return An arraylist of arraylist of string for each database item.
 * @author rokas
 */
protected static ArrayList<ArrayList<String>> searchVehicles(String searchTerm) {
    return DatabaseHandler.getData( column: "vehicles.vehicle_id, location.location_id, vehicles.type, vehicles.make, vehicles.model, vehicles.day_rate" +
                ", location.country, location.city, location.street",
            table: "vehicles, location ",  where: " vehicles.location_id = location.location_id " +
                "AND ( vehicles.type LIKE ? OR vehicles.make LIKE ? OR vehicles.model LIKE ? " +
                "OR location.country LIKE ? OR location.city LIKE ? OR location.country LIKE ? OR CAST(vehicles.vehicle_id AS TEXT) LIKE ?" +
                " OR CAST(location.location_id AS TEXT) LIKE ? OR CAST(vehicles.day_rate AS TEXT) LIKE ? )  ",
            new String[]{"%" + searchTerm + "%", "%" + searchTerm + "%", "%" + searchTerm + "%", "%" + searchTerm + "%",
                "%" + searchTerm + "%", "%" + searchTerm + "%", "%" + searchTerm + "%", "%" + searchTerm + "%", "%" + searchTerm + "%"});
}
```

The search function relies on the helper Database Handler class, which gets a connection to the database that's already established and returns the data that is given by the user inputted query. SQL inputs are sanitized by PreparedStatements which is recommended by java, this ensures all user inputs are taken as Strings preventing SQL injections.

The above SQL combines the vehicles and location table with a join and then runs a query with the user searchTerm that finds the designated columns likeness. If any of the rows in the joined table have any likeness to the search string, they will be outputted to the user in a ascii table format:

```
Enter Search Field OR ENTER "BACK"
VOLKSWAGEN
+--------------+---------------+-----------+--------------+-----------+------------+-----------+--------------------+----------------------+
| vehicle_id   | location_id   | type      | make         | model     | day_rate   | country   | city               | street               |
+--------------+---------------+-----------+--------------+-----------+------------+-----------+--------------------+----------------------+
| 8            | 4             | COUPE     | VOLKSWAGEN   | B-CLASS   | 1918       | Croatia   | Joslynborough      | Cristal Harbor       |
| 21           | 3             | SEDAN     | VOLKSWAGEN   | D-CLASS   | 1260       | Sweden    | Purdyborough       | Abbott Course        |
| 25           | 3             | SEDAN     | VOLKSWAGEN   | C-CLASS   | 1674       | Sweden    | Purdyborough       | Abbott Course        |
| 29           | 1             | COUPE     | VOLKSWAGEN   | A-CLASS   | 1400       | Spain     | Runtehaven         | Pfannerstill Vista   |
| 35           | 4             | SEDAN     | VOLKSWAGEN   | D-CLASS   | 501        | Croatia   | Joslynborough      | Cristal Harbor       |
| 40           | 8             | MINIVAN   | VOLKSWAGEN   | E-CLASS   | 1056       | Togo      | Port Jenachester   | Bruen Creek          |
+--------------+---------------+-----------+--------------+-----------+------------+-----------+--------------------+----------------------+
```

3.3.2 Add vehicles

```java
/**
 * Call TO DATABASE to add a new vehicle
 *
 * @param type        - type of vehicle
 * @param make        - make of vehicle
 * @param model       - model of vehicle
 * @param year        - year of vehicle manufacture
 * @param day_rate    - how much the vehicle costs per day
 * @param location_id - the location row where the car is first located.
 * @return true if successfully created.
 * @author Rokas
 */
static boolean addVehicleToDatabase(String type, String make, String model, String year, String day_rate, String location_id) {
    return DatabaseHandler.insertData( Table: "vehicles",  Column: "type, make, model, year, day_rate, location_id",  values: " ?, ?, ?, CAST(? as int), CAST(? as decimal), CAST(? AS int)"
            , new String[]{type, make, model, String.valueOf(year), String.valueOf(day_rate), String.valueOf(location_id)});
}
```

Adding vehicles calls another helper function to insert row, the user provides information through a long console input parsing, which allows a query call to the database to insert a row.

```java
/**
 * @return
 * @Ridwan set function for database
 */
public static boolean insertData(String Table, String Column, String values, String[] inputs) {
    PreparedStatement prepStmt = null;
    Connection c = null;
    try {
        c = DatabaseHandler.connectDatabase();
        //prepared statement against sql attacks
        prepStmt = c.prepareStatement( sql: "INSERT INTO " + Table + "(" + Column + ")" + " VALUES(" + values + ");");
        for (int i = 1; i <= inputs.length; i++) {
            prepStmt.setString(i, inputs[i - 1]);
        }
        prepStmt.executeUpdate();
        prepStmt.close();
        c.close();
    } catch (Exception e) {
        try {
            prepStmt.close();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        try {
            c.close();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        System.err.println(e.getClass().getName() + ": " + e.getMessage());
        return false;
    }
    return true;
}
```

The above helper function sanitizes the statements and ensures that SQL injections cannot be added from user input.

```java
@Test
public void canAddVehiclesToDatabase() throws UnexpectedException {
    //copy current table
    String rndm = String.valueOf((int) Math.random() * 1000);

    String oldTable = KartasticUtility.createTable(Vehicles.getDefaultVehicles(),  paddingSize: 4);
    assertTrue(Vehicles.addVehicleToDatabase(rndm,  make: "make",  model: "model",  year: "2000",  day_rate: "2.0",  location_id: "1"),  message: "query should return success");
    assertTrue(Vehicles.addVehicleToDatabase(rndm,  make: "make",  model: "model",  year: "2000",  day_rate: "2.0",  location_id: "1"),  message: "query should succeed as vehicl
    assertFalse( condition: DatabaseHandler.getData( column: "vehicle_id",  table: "vehicles",  where: " type = ?", new String[]{rndm}).size() == 2,  message: "UNIQUE type na

    //old table and new table should be equal
    assertNotEquals(oldTable, KartasticUtility.createTable(Vehicles.getDefaultVehicles(),  paddingSize: 4));
}
```

Further Junit is used to test adding vehicles, adding multiple vehicles and to ensure that Tables get updated with these vehicles.

3.3.2 Remove Vehicle

Removing vehicles is done in a similar way, user inputs are taken through a console and sent to a helper function:

```java
/**
 * Calls database to remove vehicle
 *
 * @param id - integer id of the vehicle to remove
 * @return true if vehicle was successfully removed.
 * @Author Rokas
 */
protected static boolean removeFromDatabase(int id) {
    return DatabaseHandler.deleteRow( table: "vehicles",  id_name: "vehicle_id", id);
}

/**
```

These user inputs are put into an SQL call that go through a helper function that sanitizes statements and returns true if it succeeds

```java
/**
 * Deletes a row from a table
 *
 * @param table   - table to delete from
 * @param id_name - the matching to delete from
 * @param id      - the id to delete
 * @return true if deleted successfully
 * @author rokas
 */
public static boolean deleteRow(String table, String id_name, int id) {
    Statement stmt = null;
    Connection c = null;
    try {
        c = DatabaseHandler.connectDatabase();
        //prepared statement against sql attacks
        stmt = c.createStatement();

        stmt.executeUpdate( sql: "DELETE FROM " + table + " WHERE " + id_name + " = " + id + " ;");
        stmt.close();
        c.close();
        return true;
    } catch (Exception e) {
        //failed to delete
        try {
            stmt.close();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        try {
            c.close();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        return false;
    }
}
```

This is also tested through JUnit to ensure that vehicles are able to be removed:

```
//    Given im a rental manager
//    When I remove a vehicle
//     Then the vehicle listing gets updated
@Test
public void testIntegrateRemoveVehicles() throws UnexpectedException {
    //given im a rental manager - only rental managers can access the system
    //add a vehicle
    String rndm = generateRandomString();
    String randomlocation = DatabaseHandler.getData( column: "location_id",  table: "location").get(1).get(0);
    assertTrue(Vehicles.addVehicleToDatabase(rndm,  make: "make",  model: "model",  year: "2000",  day_rate: "2.0", randomlocation),  message: "query should add new vehic
    //created vehicle exists
    assertTrue(KartasticUtility.createTable(Vehicles.getDefaultVehicles(),  paddingSize: 4).contains(rndm),  message: "the random type name exists in database");
    //store old table
    String oldTable = KartasticUtility.createTable(Vehicles.getDefaultVehicles(),  paddingSize: 4);


    //remove created vehicle by finding its status
    assertTrue(Vehicles.removeFromDatabase(Integer.parseInt(DatabaseHandler.getData( column: "vehicle_id, type",  table: "vehicles",  where: "type = ? ", new String[]
    //store new table
    String newtable = KartasticUtility.createTable(Vehicles.getDefaultVehicles(),  paddingSize: 4);
    //the vehicle is removed from the table
    assertFalse(newtable.contains(rndm),  message: "the random type name gets removed from database");
    assertNotEquals(oldTable, newtable,  message: "data is different");
}
```

4. Coding and Testing

Testing is done through Junit for each function, due to my lack of understanding, was unable to test the console inputs but was able to take functions outside of console inputs and test them which is shown through my user studies:

4.1 User Stories

This section will go over the previous user stories and look at their java implementations

4.1.2 Vehicle Search

```java
/**
 * Search for a vehicle by string
 *
 * @return true if user decided to go back
 * @Author Rokas
 */
public static boolean openVehicleSearch() {
    while (true) {
        try {
            String searchTerm = KartasticUtility.getStringInput( msg: "Enter Search Field ",  allowBack: true,  maxLength: 140,  minLength: 0);
            //search for matches
            KartasticUtility.displayTable(searchVehicles(searchTerm),  paddingSize: 4);

        } catch (InputMismatchException e) {
            //back enteredv
            return true;
        }
    }
}
```

Input uses utility class to parse in order to ensure that the correct type of input is passed to the Query.

This Utility can be seen on the next page:

```java
 * takes a string input from a user and returns the required string input.
 *
 * @param msg       - msg to display with incorrect input
 * @param allowBack - if the function should provide a back command
 * @param maxLength - int of the maximum length of string
 * @param minLength - the minimum length of string
 * @return the input that is taken;
 * @author Rokas
 */
public static String getStringInput(String msg, boolean allowBack, int maxLength, int minLength) throws InputMismatchException {

    //generate helper
    //determine if back command is given
    if (allowBack) {
        msg += "OR ENTER \"BACK\"";
    }
    msg += "%n";

    //input validation
    boolean message = false;
    System.out.printf(msg);
    String input = "";
    do {
        if (message) System.out.printf(msg);
        //provide comfort command list
        input = Main.stdin.nextLine().trim();
        //allow BACK
        if (allowBack) {
            if (input.equalsIgnoreCase( anotherString: "BACK")) {
                throw new InputMismatchException("User Entered Back");
            }
        }
        //check if input matched requirements
        if (input.length() <= maxLength && input.length() >= minLength) {
            //valid command
            return input;
        } else {
            message = true;
        }
    } while (true);
}
```

Unfortunately, there is no user testing in any of the console inputs as I could not find a way to use Junit with console Inputs reliably.
But testing is run on the Data queries and table queries

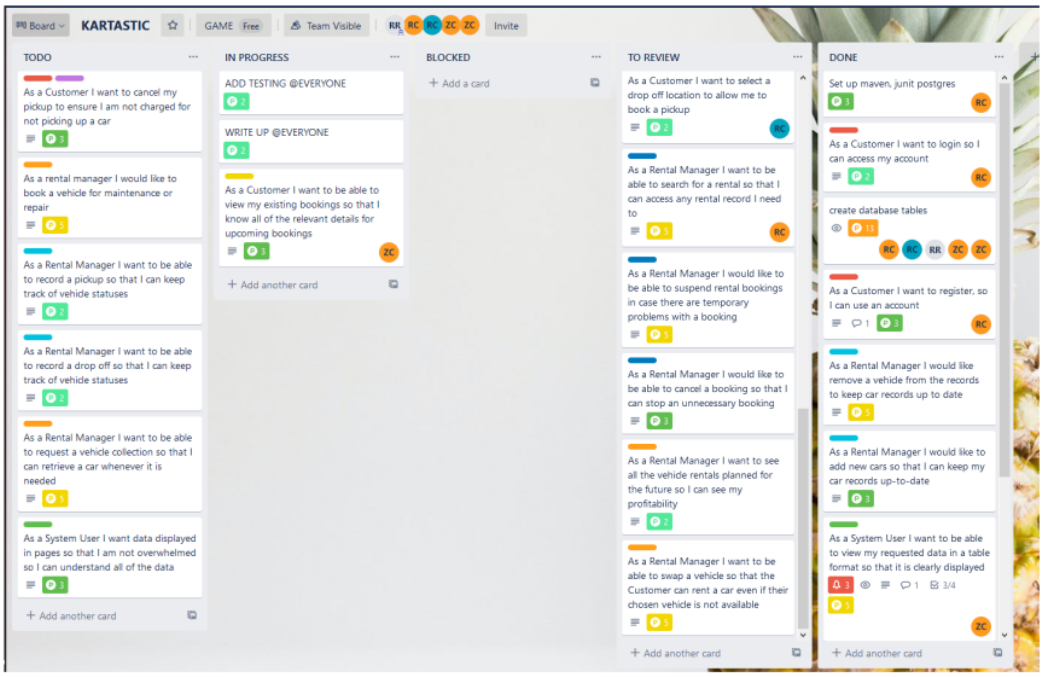
5 Reflection

Overall, this project has further developed my agile working skills and provided another opportunity to work within a team to develop my versioning tool skills. Although the entire sprint backlog/product backlog was not complete, but this is most likely due to the team taking on too much work, the final product is functional and provides a minimal viable product for customers and managers. In addition, throughout I have further developed my mentoring skills by pair programming and assisting other people with their work. In the future, I need to get better at estimating how difficult tasks are and get used to underestimating rather than overestimating so I can always produce a better result than what I have promised to do. In this project, not having a proper product owner created an interesting show case about how important they are in establishing the requirements for the product and clarifying design decisions, this makes me appreciate agile development methodologies.

When it comes to databases, the use of various SQL joins and a relational database creation has helped me put my learnt skills to the test especially in this chosen project. Although normal SQL queries were not created and instead Java JCDB was used, the combination of communicating with PostgreSQL and the application has taught a lot about importance of security on SQL injection attacks and how separation of databases and software is a necessity to provide better testing for the database.

When it comes to software development and testing, I saw how important it is to develop tests while developing functions as otherwise the project will become too big to manage. In addition, its possible to write functions to be easier and more manageable to test by simplifying large functions into smaller ones, and as such makes programs more robust. It has also allowed me to examine the project from a top-down view revealing all the code repetition and how functions can be rewritten to be more efficient and repeat less with templating and using helpers. This also makes code more robust and reduces chances of failure, through user errors.

Appendix A1



Appendix A2

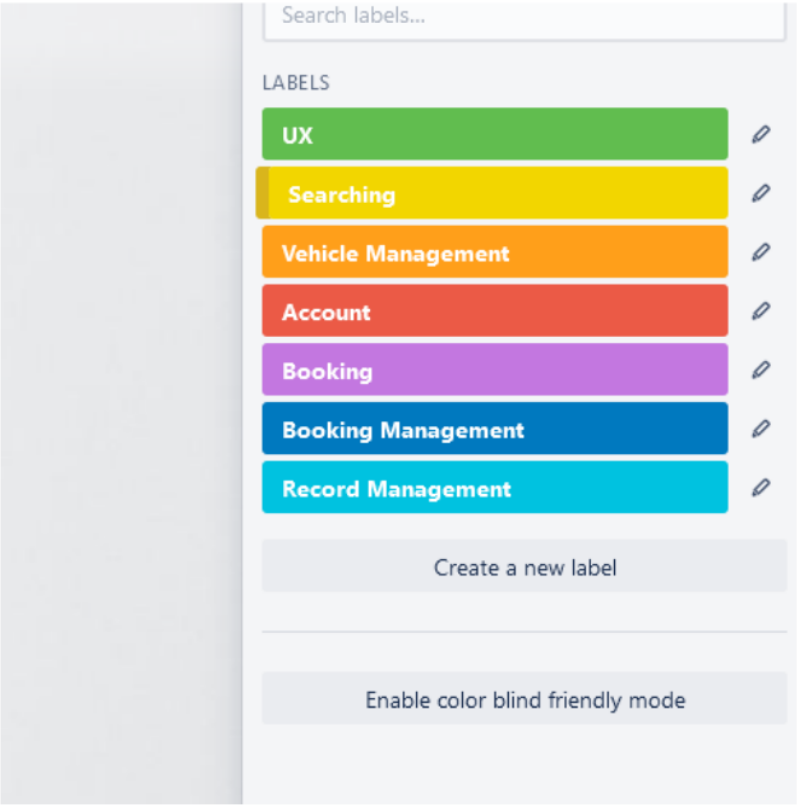
Definiton of Ready:

- Team understand enough of the architecture.
- Story must be written as a user story.
- Acceptance criteria must exist and be understood by team.
- Has no more than 3 acceptance criterias.
- Has been estimated by the team.

Definition of Done:

- The story has been fully developed.
- Push has been peer reviewed.
- The story has been fully tested.
- All tests pass.
- No bugs outstanding.

Appendix 3


Search labels...
LABELS
UX
Searching
Vehicle Management
Account
Booking
Booking Management
Record Management
Create a new label
Enable color blind friendly mode

Kartastic Report

GRADEMARK REPORT

FINAL GRADE

82/100

GENERAL COMMENTS

Instructor

Software design evaluation (30): 26

Database evaluation and implementation (30): 24

Coding and Testing (30): 23

Reflection (10): 9

A good, cohesive report which has clearly been well-structure and considered.

It would have been nice to see a more detailed overview of the user story map and discussion of the overarching design process therein.

Some good reflective comments in each section.

A well-written and fully featured solution to the problem. The code is clean, though would benefit from a few more comments in places. The structure of the code is logical and well-considered.

The vehicle class appears to be well tested, but the remaining code lacks coverage.

Clear representation of the database requirements represented through the ERD. Be careful to test for data duplication, there's some rationalisation possible around rentals/payments. Otherwise a good solution.

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

PAGE 19