**Portfolio Milestone One**

Michael Maye

Colorado State University Global

CSC470: Software Engineering

Dr. Shaher Daoud

07/30/2023

**Portfolio Milestone One**

.

**Overview**

The submitted milestone explores a draft of a database model that investigates object-oriented database design and integration with relational database mapping. Throughout the design phase of a system, it is crucial to iterate through different approaches to explore the advantages and disadvantages of using the respective iterations. The analysis of this submission will explore different iterations of database structure for three classes, Vehicle, Car, and Truck, along with their effects on object class structure.

**Analysis**

The following analysis will explore the creation of three classes, Vehicle, Car, and Truck, each with two attributes. These classes will be foundational in drafting a relational table system that will store the objects and iterate through single, two, and triple-table formats. The two-table format will then assist in recreating the car and truck objects. The reformatting of 'Car' will explore the different relational table formats and their impact on class design. Finally, it will introduce a Driver class and explore different multiplicities and associations between the Driver and Car class.

**Initial Classes**

The initial creation of the systems classes included modeling a 'Vehicle', 'Car', and 'Truck' class. The 'Vehicle' class is the parent class which holds the public attributes 'vehicleID' and 'color'. The 'Car' and 'Truck' classes inherit the attributes from 'Vehicle' and hold their own pubic attributes. 'Car' holds 'make' and 'model', which are strings related to the car's manufacture and model tag, respectively. 'Truck' holds 'payloadWeightLimit', the weight limit of contents put in the truck's bed, and 'truckBedSizeSqFt', the square foot size of the truck's bed.

**Figure 1**

A diagram of a vehicle

Description automatically generatedVehicle, Car, Truck class structure

**Database Tables**

**Single Table**

The single table approach to database design involved taking all required data and creating a single table to hold them all. As 'Vehicle' is the parent, the primary The single table approach to database design involved taking all required data and creating a single table to hold them all. As 'Vehicle' is the parent, the primary attribute in this table will be 'vehicleID', which will map to each 'Vehicle' object regardless of class type. Each vehicle class object will have a color, as they inherit it, but will have several attributes set to 'null'. For example, a 'Car' object will have 'make' and 'model' but will not have a 'truckBedSizeSqFt' attribute, so for that row, it will be set to null (figure 1). This database structure combines all data into one table. It allows the system to query one database instead of querying two or three tables to return an object’s data. With only two classes, it is easy to read. However, if the system includes other vehicle classes, the table can become overwhelming and difficult to query for a single return.

**Figure 2.1**

Single Table structure

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **vehicleID** | **color** | **make** | **model** | **payloadWeightLimit** | **truckBedSizeSqFt** |
| 1 | Red | Honda | Civic | NULL | NULL |
| 2 | Blue | Toyota | Corolla | NULL | NULL |
| 3 | Black | NULL | NULL | 5000 | 54 |
| 4 | Tan | NULL | NULL | 7500 | 75 |

**Two-Table**

With a two-table approach, the single table splits into two different tables, one for ‘Car’ and one for ‘Truck’. With the tables split, the system can continue identifying each ‘Vehicle’ with a ‘vehicleID’ and ‘color’ attribute while avoiding using ‘NULL’ for attributes that do not exist on that vehicle type. This table structure gives one table for ‘Car’ that includes its ‘Vehicle’ attributes and its ‘Car’ attributes and a table for ‘Truck’ that includes both ‘Vehicle' and 'Truck’ attributes. As this particular system has only two classes that inherit from ‘Car’, this creates a structure that holds all data for system objects while avoiding ‘NULL’ entries with the smallest number of tables..

**Figure 2.2**

Two-Table structure

***CAR:***

|  |  |  |  |
| --- | --- | --- | --- |
| **vehicleID** | **color** | **make** | **model** |
| 1 | Red | Honda | Civic |
| 2 | Blue | Toyota | Corolla |

***TRUCK:***

|  |  |  |  |
| --- | --- | --- | --- |
| **vehicleID** | **color** | **payloadWeightLimit** | **truckBedSizeSqFt** |
| 3 | Black | 5000 | 54 |
| 4 | Tan | 7500 | 75 |

**Three-Table**

The three-table approach will divide the storage responsibility between the previous two tables by pulling the ‘Vehicle’ attributes out and pushing them into their own tables. The ‘Vehicle’ table will hold the ‘vehicleID’ of each object and map it to that object's ‘color’ attribute. The objects ‘vehicleID’ will then be used in the ‘Car’ and ‘Truck’ tables, respectively, to map each object type to the class attributes of each. This can be incredibly helpful when the parent class has many attributes overwhelming the child class tables. This system only has one parent attribute that will not be used in subsequent tables, so using a three-table structure is difficult to justify.

**Figure 2.3**

Three-Table structure

***VEHICLE:***

|  |  |
| --- | --- |
| **vehicleID** | **color** |
| 1 | Red |
| 2 | Blue |
| 3 | Black |
| 4 | Tan |

***CAR:***

|  |  |  |
| --- | --- | --- |
| **vehicleID** | **make** | **model** |
| 1 | Honda | Civic |
| 2 | Toyota | Corolla |

***TRUCK:***

|  |  |  |
| --- | --- | --- |
| **vehicleID** | **payloadWeightLimit** | **truckBedSizeSqFt** |
| 3 | 5000 | 54 |
| 4 | 7500 | 75 |

**Car and Truck reformatting**

The Car and Truck class diagrams can now be reformatted using the two-table structure. According to the tables in the two-table structure, the ‘Car’ class contains the ‘vehicleID’, ‘color’, ‘make’, and ‘model’ attributes and will need to reflect this in its class diagram. In the same way, the ‘Truck’ class diagram will be updated with the ‘vehicleID’ and ‘color’ attributes. The final class diagrams (figure 3) contain the inherited ‘Vehicle’ attributes and those from their class type. With this structure, we can abstract away from using the ‘Vehicle’ class and only store data from the ‘Car’ and ‘Truck’ classes.

**Figure 3**

Car and Truck class diagrams

A diagram of a vehicle

Description automatically generated

**Car Driver Association**

To begin, the Driver class will be associated with the Car class with a multiplicity of 1 on the Driver side and N on the Car side (figure 4.1). This association indicates that drivers can have multiple cars, but cars can only have one driver. A table holding Cars and Drivers would include each driver and the cars they are associated with (figure 4.2). Updating the multiplicity from 1 to N (figure 4.3) indicates that both Cars and Drivers can be associated with any number of the other, and an updated table shows that there are both multiple cars for each driver and multiple drivers for each car (figure 4.4).

**Figure 4.1**

1 to N Class diagram

A diagram of a car

Description automatically generated

**Figure 4.2**

1 to N Database Table

***CAR:***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **vehicleID** | **color** | **make** | **model** | **driverID** |
| 1 | Red | Honda | Civic | 1 |
| 2 | Blue | Toyota | Corolla | 1 |
| 3 | White | Chevy | Volt | 2 |
| 4 | Yellow | Subaru | Outback | 2 |

***DRIVER:***

|  |  |
| --- | --- |
| driverID | name |
| 1 | Scott |
| 2 | Frank |
| 3 | Mary |
| 4 | Tom |

**Figure 4.3**

N to N Class Diagram

A diagram of a computer

Description automatically generated

**Figure 4.4**

N to N Database Table

***CAR:***

|  |  |  |  |
| --- | --- | --- | --- |
| **vehicleID** | **color** | **make** | **model** |
| 1 | Red | Honda | Civic |
| 2 | Blue | Toyota | Corolla |
| 3 | White | Chevy | Volt |
| 4 | Yellow | Subaru | Outback |

***DRIVER:***

|  |  |
| --- | --- |
| driverID | name |
| 1 | Scott |
| 2 | Frank |
| 3 | Mary |
| 4 | Tom |

***CAR-DRIVER:***

|  |  |
| --- | --- |
| **vehicleID** | **driverID** |
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 2 | 2 |

**Conclusion**

The submitted analysis has examined the advantages and limitations of various database structures for object storage. The mapping of the ‘Vehicle’, ‘Car’, and ‘Truck’ classes showed the complexities and considerations involved in designing a database to decompose and store objects. The addition of the ‘Driver’ class and its association with the ‘Car’ class also introduced another layer of complexity where the database was able to properly store the ‘driverID’ in the original ‘Car’ table but required the addition of a separate table to map different drivers to different cars once the multiplicity was N to N between ‘Car’ and ‘Driver’. Exploring different approaches and structures in this way allows for a complete understanding of the requirements and performance of different structures. It leads developers to properly design the most efficient and effective database for the system.

**References**

Class diagrams. in UML modeling. (n.d.). https://www.ibm.com/docs/en/rsm/7.5.0?topic=structure-class-diagrams

IBM developer. (n.d.). https://developer.ibm.com/articles/the-class-diagram/

UML class diagram - javatpoint. www.javatpoint.com. (n.d.). https://www.javatpoint.com/uml-class-diagram

Unhelkar, B. (2020). 13. In Software engineering with UML. essay, CRC PRESS.