Software Engineering Portfolio Project

Alexander Ricciardi

Colorado State University Global

CSC470: Software Engineering

Dr. Vanessa Cooper

February 9, 2025

Contents

Software Engineering Portfolio Project

This document is my final portfolio project from CSC470 Software Engineering winter B semester at Colorado State University Global (CSU Global). The project is composed of three parts as follow:

Option #1: **Database Model**

The Portfolio Project, Option #1, consists of three parts. Please review all three and submit together for grading.

Note: Part 1 was completed as a draft in the Module 7 Portfolio Milestone, Option #1. Please make any updates necessary to that Milestone, being sure to incorporate instructor feedback.

Part 1: Database Model

Create a database model while following the steps listed below:

- Create a simple class diagram containing three classes: Vehicle, Car, and Truck. Provide two attributes for each of these three classes.
    - Create a rough sketch of a relational table that will store the objects belonging to the aforementioned three classes.
    - Ensure you have tried all three options—single table, two tables, and three tables—in your database design.
    - Enter two objects per table, corresponding to the classes.
- Use the two-table design to recreate a Car object.
- Repeat the preceding step to recreate a Truck object.
- Create a simple association relationship in a separate class diagram showing Driver and Car.
    - Apply a multiplicity of 1 on the driver side and N on the Car side.

- Add/modify your table designs to handle storing of two objects belonging to Car and two belonging to Driver.
    - Modify the multiplicity on the Driver side to N. This makes it a many-to-many multiplicity.
- Modify your table designs to handle this multiplicity and show where and how the KEYS or IDs will have to be placed.
- Create a class diagram corresponding to the tables you have designed. Stereotype all classes on that class diagram as <<table>>.

Part 2: Pseudocode

Using a programming language of your choice, write a script that will print out pseudocode of your model. Paste the pseudocode in a Word document.

Part 3: Lessons Learned Reflection

Write a one- to two-page summary that outlines the lessons you have learned in this software engineering course. Reflect on how these lessons can be applied toward more effective software engineering practices and modeling.

Compile all your work (Parts 1-3) into a single document and submit as a Word or PDF file.

Your paper must be formatted according to the APA guidelines in the CSU Global Writing Center.

CSU Global (n.d.)

Project Part-1

The first part of this project is an essay titled: “Object-Relational Mapping: Designing a Vehicle Database Management System with UML Class Diagrams.” The essay explores the concepts of Object-Oriented Programming (OOP) in software development, mapping objects (classes) to the schema of the widely used Relational Database Management Systems (RDBMS) using Object-Relational Mapping (ORM). To demonstrate these principles of data modeling, including ORM and CRUD operations, the essay provides a vehicle database management system example, supported by a UML class diagrams.

Project Part-2

The second part of this project provides the pseudocode of a vehicle database management system based on the final class diagram from the essay in part-1 of this project and PlantUML code. The pseudocode was generated using a small Java program named: “PlantUML Class Diagram To Pseudocode Generator Version-5”. The program translates PlantUML class diagrams to pseudocode by parsing the PlantUML code using Regex and Java methods. It also generates comments based on CRUD operations. I created this program to meet part-2 requirements. Additionally, this document provides the program Java code and all the material within this document can be found in the following GitHub repository: https://github.com/Omegapy/My-Academics-Portfolio/tree/main/Software-Engineering-CSC470/Module-8-Portfolio-Project

Project Part-3

The third part of this project provides a reflection essay about the lessons that I have learned in the CSC470 Software Engineering course, and how they can be applied to more effective software engineering practices and modeling.

Part-1 - Object-Relational Mapping: Designing a Vehicle Database Management System with UML Class Diagrams

Applications, from e-commerce platforms and social media networks to financial systems, AI development, scientific research, and gaming, depend on their ability to store, manage, and analyze data to function properly and reach their goals. Databases allow data to be stored persistently and structured in a way that can be retrieved reliably for use, manipulation, and analysis. In Software Engineering, data modeling is essential for understanding and developing the relationships between a system's data, its requirements, and its functionality. Given the prevalence of Object-Oriented Programming (OOP) in software development, understanding how to map objects (classes) to the schema of the widely used Relational Database Management Systems (RDBMS) using Object-Relational Mapping (ORM) is essential. To demonstrate the principles of data modeling, including ORM and CRUD operations, this essay provides a vehicle database management system example, supported by a UML class diagrams.

Databases Modeling Overview

In Software Engineering (SE), “data modeling is the process of creating a visual representation of either a whole information system or parts of it to communicate connections between data points and structures” (IBM, n.d.a, p.1). In other words, the main goal of data modeling is to illustrate the types of persistent data used and stored by a system and the relationships between the persistent data types and the different components of the system. Persistent data are objects that continue to exist beyond a single execution of a computer program (Unhelkar, 2018). Objects in a computer system’s memory are transient meaning that they exist only during the execution of the system program. Data persistence enables a system to store objects at the end of its program executions and retrieve them when a new instance of the program execution begins. This data can be stored in the form of Object-Oriented Databases (OODBs) also called Object Database Management Systems (OODBMS), NoSQL databases, and Relational Databases (RBs) also called RDBMS. Modeling databases is as challenging as modeling business logic; thus it is crucial for software engineers to understand how Object-Oriented concepts (OO) relate to database modeling as they often need to map OO concepts like classes, objects, inheritance, and polymorphism used in OOP to database data types and structures.

Object-Oriented Concepts Overview

In SE, an object can be defined as an entity that encapsulates internal processes and provides an interface for external processes to interact with it (Galton & Mizoguchi, 2009). Object-Oriented (OO) concepts build upon this, forming a paradigm where systems and real-world entities (e.g., users) are defined as "objects" which are abstractions that encapsulate both data (attributes) and operations performed on that data (methods) (Ricciardi, 2025a). In the context of OOP, the data represent the attributes that define the characteristics of a class, while the operations on the data represent the methods that define the behavior of an instantiated object from the class (Colorado State University Global, n.d.). In other words, classes can contain both executable code in the form of methods and data in the form of attributes. In essence, OOP embodies the principles of OO in blueprints called classes, where those concepts are implemented in specific ways. The list below lists the main principles of OO.

- Classification ⟛Type of object⟛
- Abstraction ⟛ Hiding the object's internal implementation details, including some of its attributes and the functionalities of its methods, while representing those attributes and methods internally ⟛

- Encapsulation —Defining the boundary of an object (modularization), defining an object’s attributes and methods—
- Association —Relationships defining objects' interactions with each other—
- Inheritance —Relationships defining objects as a generalization (parents) of other objects (children)—
- Polymorphism —How objects respond differently to the same message (same method call)—

(Ricciardi, 2025a, p.1)

To store and manipulate objects in the form of persistent data, developers use various database types, including OODBMS, which are well-suited to directly storing objects without the need for type mapping (modifying the object schema to fit a non-object-oriented data schema).

To summarize, OO concepts, including classification, abstraction, encapsulation, association, inheritance, and polymorphism, are essential concepts of SE. These concepts are used to design systems where real-world entities are defined as objects that encapsulate both data (attributes) and the operations (methods) performed on that data. Therefore, understanding OO concepts is critical when mapping object-oriented designs to databases.

Object-Oriented Database Overview

OODBMS are based on the OO paradigm storing data as objects with their attribute values, operations, and relationships (Unhelkar, 2018). OODBMS stores the object data and structure “as-they-are” preserving the objects’ relationships like inheritance, association, and aggregation. This facilitates greatly the design of OO-based software systems, as objects can be retrieved from the databases and loaded directly into a computer memory system without adding extra processes for data translation. OODBMS can also directly store “as-is” complex data types such as Binary Large Objects (BLOBs) and unstructured data (e.g., video and audio) making them useful for retrieving, storing, manipulating these data types within applications that require retrieval, storage, and manipulation of such data, for example, multimedia systems and scientific data management platforms. Besides storing objects and complex data, OODBMS have several other advantages as well as disadvantages. Table 1 provides a list of some of these advantages and disadvantages.

Table 1

OODBMS Advantages and Disadvantages

Advantages	Disadvantages
Can directly store object storage: - Integration with OOP - Objects (data, methods, relationships) are stored "as is," eliminating assembly/disassembly. - Supports BLOBs and complex unstructured data without format conversion.	Has performance limitations: - Lower efficiency for simple data/relationships. - Possible slower access due to late binding. - Poor performance for simple queries compared to Relational Database Management Systems (RDBMS). - Not well-suited for dynamic data that is constantly changing, as changes to an object will require to rewrite the entire object.
Can handle complex data: - Supports inheritance, polymorphism, encapsulation, and complex data types (e.g., spatial, multimedia). - Easier navigation (search) through object hierarchies.	It is difficult to adopt and support: - Limited adoption and developer expertise. - Fewer user tools, libraries, and community support compared to RDBMS.
Fast development and implementation**:** - No need to map objects to tables (simpler mapping). - Less code is required for object-oriented applications.	No standardization: - Lack of standardization across vendors. - Less stable standards than RDBMS (risk of changes).
Can model the real world:	Scalability challenges:

- Data model aligns with real-world entities and OOP principles. - Relationships (inheritance, association) are stored directly.	- Complex data models may hinder partitioning data across nodes. - Vertical/horizontal scaling requires specialized infrastructure.
Well-suited for concurrency control encapsulation: - Better concurrency control (hierarchy locking). - Encapsulation improves data security and integrity.	High cost and difficulty to integrate with other systems: - Higher costs (specialized software/hardware). - Difficult integration with BI/reporting tools.
Suited for distributed architecture: - well-suited for applications that run across multiple computer systems and systems connected over a network (i.e., a distributed system)	Can be complex: - Object-oriented paradigms can be more complex than relational models. - Steeper learning curve for developers accustomed to relational models.

Note: The table lists the advantages and disadvantages of Object Database Management Systems (OODBMS). From "Object-Oriented Principles in Software Engineering: An Overview of OODBMS, RDBMS, and ORM Techniques" by Ricciardi (2025a).

To summarize, OODBMS stores data as objects with their relationships and structure, making them ideal for object-oriented applications. Additionally, they directly store complex data types making them well-suited to store and manage BLOBs and unstructured data. However, they also have disadvantages like low performance for simple queries and a lack of standardization.

NoSQL Databases Overview

NoSQL databases are databases that enable the storage and querying of data outside the traditional structures found in relational databases (data stored in tables) (IBM, 2022b). They can still handle relational data (tables) storing it differently than RDBMS does. Additionaly they can accommodate unstructured data such as unstructured email or feedback on social media (Unhelkar, 2018). They are well-suited to handle large-scale data due to their technology, federated database structure, and their adaptable format schemas. Federated databases are composed of multiple connected databases forming a single unified network (Navlaniwesr, 2024). A real-world example of a federated database could be a data system used by a multinational corporation that has offices in multiple countries (Rajpurohit, 2023). In addition to being capable of handling relational and unstructured data, and well-suited for federated database structure, NoSQL databases have other advantages as well as disadvantages. Table 2 provides a list of some of these advantages and disadvantages.

Table 2

NoSQL Databases Advantages and Disadvantages

Advantage	Disadvantage
- Scalable. - The data can be distributed across several servers making the system highly scalable. - Handles large datasets and high traffic, - Adaptable.	- Require extra infrastructure, increasing costs and overhead. - Scalability is not automatic and it is based on the specified data used by the system,
- Good performance. - Can be optimized for various data types and access needs. - More performant than SQL databases in queries single large database.	- For complex joins and ad-hoc queries., it is less performant.
- flexible schema. - Has a flexible or less schema making it easy to store unstructured data and evolving data structures, - No need to predefine the data structure.	- Has security challenges due to its flexible schemas and distributed nature.
- Handles various data types. - Accommodate structured, semi-structured, and unstructured data including text, images, videos, documents, etc, - Can handle various formats like JSON, XML, etc.	- Can have consistency issues when data updates are not immediately implemented across all nodes.
- Has configurable consistency levels for optimizing performance needs.	- Minimal support for ACID (Atomicity, Consistency, Isolation, and Durability).

- Complies with GDPR, CCPA, and HIPAA regulations,	- Its distributed nature and flexible schema make the implementation of security measures challenging.
- Cost effective, - Open-source in most cases, reducing licensing costs.	- Lacks standardization - NoSQL databases lack standardization, often leading to vendor lock-in.
- Handles various data structures - Stores and manages unstructured data "as is". - Can handle federated database structure.	- Evolving data structures such as document databases can make the system hard to manage.
- Ideal for extremely large datasets management, - Ideal for content management, social media, IoT, and big data analytics.	- Not ideal for applications needing strict transactional guarantees (e.g., financial systems),
- Often cloud-implemented. - Many are cloud-native, making them more scalable, cost-efficient, easier to manage.	

Note: The table lists the advantages and disadvantages of No-SQL databases. Data from several sources (Unhelkar, 2018; ScyllaDB, n.d; InterSytems, n.d.b; Demarest, 2025; Adservio, 2021; Imperva n.d.; Foote, 2022, Quach, n.d.; Macrometa, n.d.; MongoDB, n.d., InterSystems n,d. a)

To summarize, NoSQL databases allow for flexible storage and querying of data, including both relational and unstructured data. They are scalable, performant, and adaptable making them suitable for large datasets and federated database systems. However, they also have disadvantages like consistency, security issues, and low standardization.

Relational Database Management Systems Overview

RDBMS are databases that store data in the form of tables representing predefined relationships through rows and columns (Microsoft, n.d.; Google n.d.). These relationships in RDBMS are established through logical connections, typically using primary and foreign keys allowing for querying and management of the data using programming languages such as Structured Query Language (SQL) (Ricciardi, 2025a). A primary key is a unique identifier for each record (row) in a table, while a foreign key in one table references the primary key of another table, creating a link between them (InterSystems, n.d.a). RDBMS is ideal for storing, retrieving, and managing data that are structured and can be placed in rows and columns (Unhelkar, 2018). Although ODBMS are better suited to handle object-based data and NoSQL databases are more flexible than RDBMS, most commercial business applications still use relational databases as the technology associated with them is affordable, reliable, standardized, widely available, and supported by a vast array of tools. In addition to these advantages, RDBMS have several other advantages as well as disadvantages. Table 3 provides a list of some of these advantages and disadvantages.

Table 3

RDBMS Advantages and Disadvantages

Advantages	Disadvantages
Simplicity: - Simplicity of the relational model. - Easier to understand, implement, and manage.	Scalability: - Performance can degrade with extremely large datasets and high transaction volumes. - Features like transactions and joins become less efficient across multiple servers.
Accuracy and data integrity: - Primary and foreign keys prevent duplicate data. - Enforces data accuracy - Data typing and validity checks ensure correct data input. - Warns when data is missing.	Schema: - Structure is defined beforehand (fixed schema), making changes difficult and potentially leading to downtime. - Adding new columns or altering the schema often requires modifying existing applications. - Inflexible for frequently changing data requirements.
Security: - Role-based security limits data access.	Cost: - Software for creating and configuring a relational database can be expensive. - Managing the database often requires specialized expertise.

Accessibility and flexibility: - Multiple users can access data simultaneously; Easy to navigate and query tables using SQL. - Easy to add, update, or delete tables, relationships, and data without impacting the overall database or applications.	Performance: - Can be slower than other database types, especially with large datasets or complex queries.
ACID compliance: - Supports Atomicity, Consistency, Isolation, and Durability for reliable transactions. - Atomicity: Defines all the elements that make up a complete database transaction. - Consistency: Defines the rules for maintaining data points in a correct state after a transaction. - Isolation: Keeps the effect of a transaction invisible to others until it is committed. - Durability: Ensures that data changes become permanent once the transaction is committed.	Memory: - Can occupy a significant amount of physical memory due to its tabular structure.
Ease of use and collaboration: - Complex queries can be easily run using SQL. - Even non-technical users can learn to interact with the database - Multiple users can operate and access data concurrently. - Built-in locking prevents simultaneous access during updates.	Object-Orientation: - Does not inherently support object-oriented concepts like classes and inheritance - Challenging to represent certain types of data or relationships that are better suited for object-oriented models.
Database design: - Reduces data redundancy and improves data integrity (Normalization). - Supports one-to-one, one-to-many, and many-to-many relationships.	Database design: - Designing a schema can be complex and time-consuming, requiring significant expertise, especially for large applications. - Proper normalization and establishing relationships can be time-consuming and require expertise.

Note: The table lists the advantages and disadvantages of Relational Databases Management Systems (RDBMS). From “Object-Oriented Principles in Software Engineering: An Overview of OODBMS, RDBMS, and ORM Techniques” by Ricciardi (2025a).

To summarize, RDBMS stores data in tables with predefined relationships through rows and columns and uses SQL for data management. They have advantages like simplicity, accuracy, data integrity, security, and accessibility, making them more widely used than the other database types. However, they also have disadvantages like difficulty to scale with extremely large datasets and a rigid schema that can be difficult to modify.

Mapping Objects to RDBMS

As mentioned previously, RDBMS are widely used; however, most software applications are designed using OOP principles implying that persistent object-based data, in systems with RDBMS backends, must be mapped to a relational schema composed of tables, rows, and columns. ORM is an RDBMS concept that helps bridge the gap between OOP and relational databases (Rajputtzdb, 2024). In ORM, classes are mapped to tables, the class attributes are matted to the columns of the corresponding tables, and the instantiated class object to rows of the corresponding table (Ricciardi, 2025a). The next section explores ORM concepts and database modeling through a vehicle database management system example.

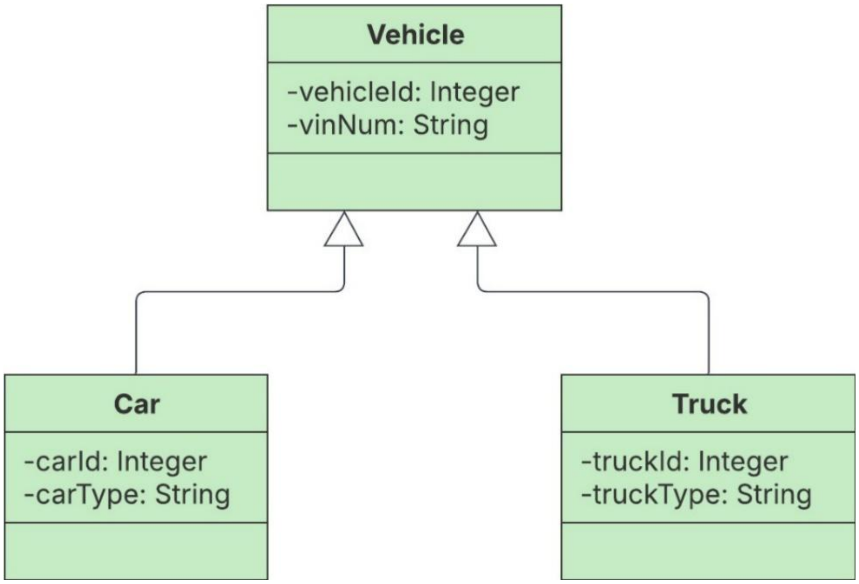
Vehicle Database Management System Using Relational Database

This section explores step-by-step the modeling of a vehicle database management system using RDMS. Starting with a simple class diagram containing three classes: *Vehicle*, *Car*, and *Truck* with two attributes. Then the classes (represented by the class diagram) are mapped, using OMR concepts, to a single-table design, a two-table design, and a three-table design. In the next step, using the two-table design the *Car* and *Truck* tables are populated with object data. Next, another class diagram is created showing a *Driver* and *Car* association, first with a one-to-many and then a many-to-many relationship, and the table models are modified to reflect the different multiplicities using primary and foreign keys. Finally, a class diagram is created representing the final RDMS design, using the <<*table*>> stereotype, as well as the corresponding relational tables.

Step-1 Diagram for Vehicle, Car, and Truck

Figure 1

Initial Vehicle System Class Diagram

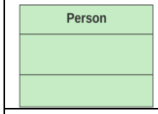
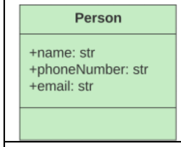
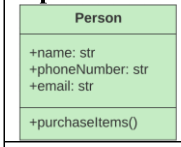
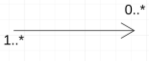
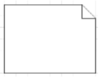
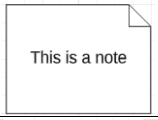
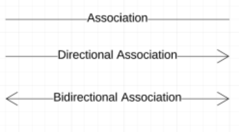
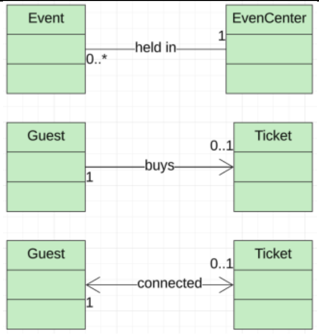
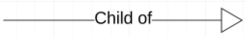
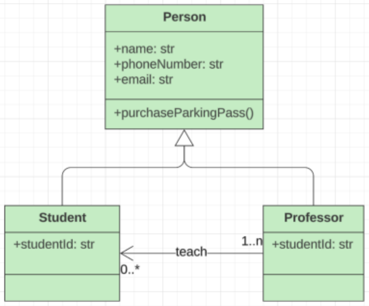
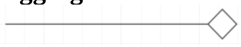
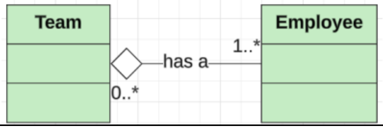


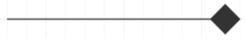
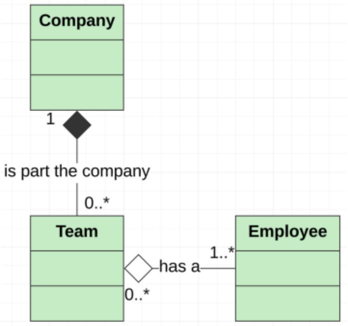
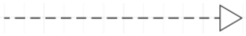
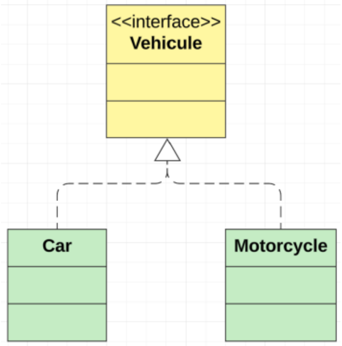
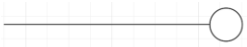
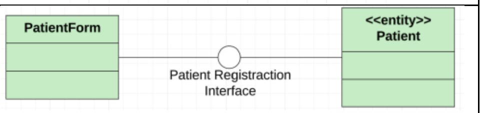
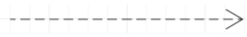
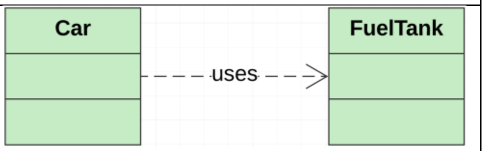
Note: The figure illustrates the initial vehicle system class diagram used to design the single-table design, a two-table design, and a three-table design. The diagram was made using the Lucid App.

In SE, Unified Modeling Language (UML) class diagrams are a type of structural diagram used to model object processes and the static structures of a system and its subsystems (Ricciardi, 2025b). They also model and illustrate class-to-class relationships such as associations, inheritance, aggregation, composition, dependency, and realization. Table 4, lists the different notation elements that can be found in a class diagram. The table also provides definitions of class diagram components including class-to-class relationships, as well as examples of usage of those components and notations.

Table 4

UML Class Diagram Notation

Component	Definition	Example:
Class  	Represents an object or objects that share a common structure and behavior.	Person, Doctor, Department.
Attributes  	Represent properties of a class that describe a range of data values that instances of the class may hold.	+name: str +phoneNumber: str +email: str
Operations  Person +name: str +phoneNumber: str +email: str +purchaseItems()	Represent Functions or methods that can be applied to or by objects of a class.	+purchaseItems()
Multiplicity  	Indicates the number of instances of one class linked to one instance of another class.	1..*, 0..*
Notes 	Represents annotations, it is used to add comments or explanations to a diagram.	
Association  Association Directional Association Bidirectional Association	Represent a relationship between two classes where objects of one class are connected to or use the services of objects of another class.	 Event EvenCenter held in 1 0..* Guest Ticket 0..1 buys 1 Guest Ticket 0..1 connected 1
Inheritance  	Represents a relationship where one class (subclass) inherits the attributes, operations, and relationships of another class (superclass). Also known as generalization.	 Person +name: str +phoneNumber: str +email: str +purchaseParkingPass() Student +studentId: str Professor +studentId: str teach 1..n 0..*
Aggregation 	Represent a specialized form of association representing a "whole-part" or "has-a" relationship. Also referred to as	

	"by reference" implying that the relationship is only a reference (pointer) to an object	
Composition 	Represent a stronger form of aggregation where the "part" cannot exist independently of the "whole."	 Company 1 is part the company 0..* Team has a 0..* 1..* Employee
Realization 	Represents a relationship between an interface and the class that realizes or implements it.	
Interface 	Represents a collection of operation signatures without implementation details.	 PatientForm Patient Registraction Interface <<entity>> PatientThe "Patient" class is a substantial class containing numerous operations. However, not all of these operations are needed by the "PatientForm" class, it uses only one of the "Patient" interfaces, the "Patient Registration Interface" subclass.
Dependency 	Represents a relationship indicating that one class (the client) depends on another class (the supplier). Changes in the supplier may affect the client.	

Note: The table provides a list of the different notation elements that can be found in a class diagram, definitions of class diagram components, and examples of usage of those components and notations. From "UML Class Diagrams: Modeling Systems from Problem Space to Solution Space" by Ricciadi (2025b). Modified.

Figure 1 illustrates the initial vehicle system class diagram, depicting the *Vehicle*, "*Car*", and *Truck* with two attributes each. The diagram also shows that the *Car* and *Truck* are subclasses of the superclass *Vehicle*, meaning that the *Car* and *Truck* classes inherit the attributes and methods of the superclass *Vehicle.* This plays a significant on how the attributes of the *Vehicle* are going to be mapped within the *Car* and *Truck* classes. Additionaly, the 'id' (e,g. *carId*) attributes of each class play significant roles, as keys, in the mapping process.

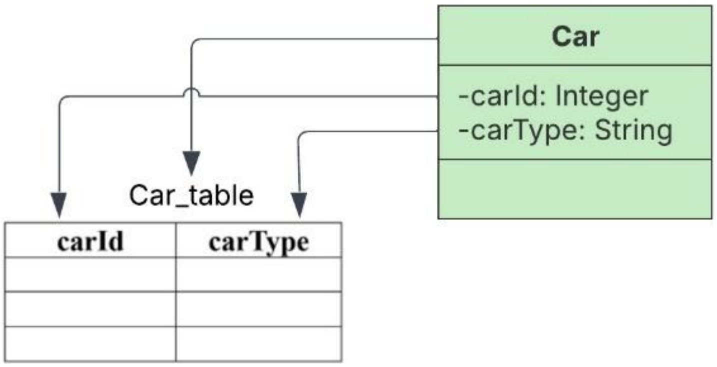
To summarize this step, Table 1 provides illustrations and definitions of the main UML class diagram notations (Class, Attributes, Operations, Multiplicity, Association, Inheritance, etc.). Class diagrams are used to model the static aspects of a system, and the class diagram in Figure 1 depicts a set of classes illustrating a vehicle system having inheritance relationships, where subclasses inherit attributes and methods from the superclass. Moreover, the id attributes represented within each class are crucial for mapping those classes to relational databases.

Step-2 Exploring the Single-Table, Two-Table, and Three-table designs

One of the simplest ORM techniques is one-to-one mapping, "where classes are mapped to tables, class attributes are mapped to the table columns, and the instantiated objects are mapped to rows in the corresponding table" (Ricciardi, 2025b, p. 2). Figure 2 illustrates how this is done.

Figure 2

ORM One-To-One Mapping Technique

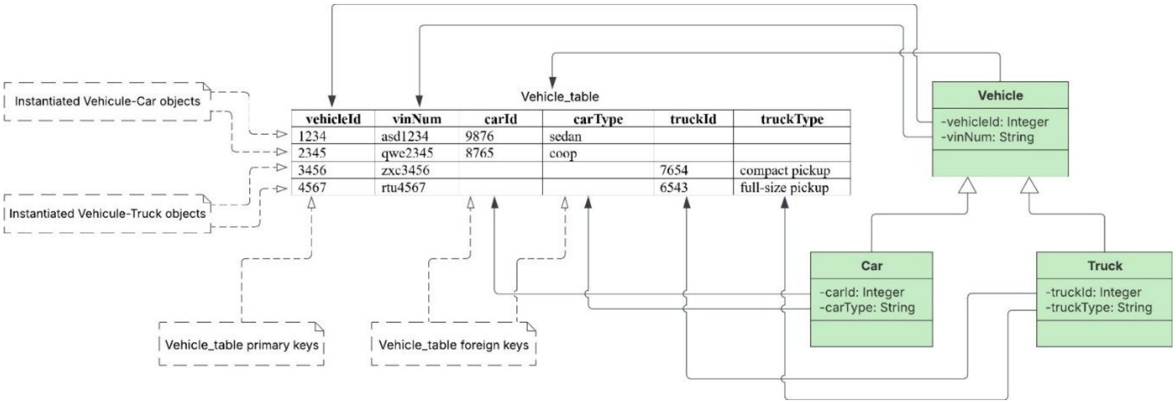


Note: The figure illustrates the ORM one-to-one mapping technique of the "*Car*" class. Mapping the *Car* class to the *Car_table*.

As shown in Figure 2, the attributes of the *Car* class are mapped to the columns of the *Car_table* table. However, this technique does not map to the *Car* class its attributes are inherited from the superclass *Vehicle.* Nonetheless, the ORM one-to-one mapping technique is the first good step in modeling class-to-table. To fully represent the inheritance relationship between the classes, a better approach is needed, such as mapping the classes to single-table, two-table, or three-table designs.

Figure 3

The Single-Table Design



vehicleId	vinNum	carId	carType	truckId	truckType
1234	asd1234	9876	sedan		
2345	qwe2345	8765	coop		
3456	zxc3456			7654	compact pickup
4567	rtu4567			6543	full-size pickup

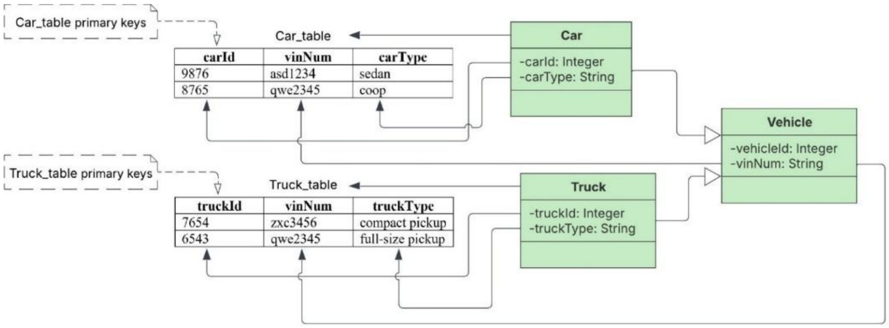
Note: The figure illustrates the mapping of the superclass *Vehicle* to the single-table design *Vehicle_table*.

Single-Table Design

In the single-table design illustrated in Figure 3, all the attributes from the superclass and the subclasses are mapped to the columns of the *Vehicle_table* table. This table stores all the objects that can be categorized as instantiated *Vehicle* objects, which include the *Car* and *Truck* objects due to their inheritance from the *Vehicle* class. Note that the row represented data saved from instantiated *Car* and *Truck* objects. In the context of RDMS, tables are also called relations, this table can be described as the inheritance table storing data from the *Vehicle* superclass attributes and its subclasses, *Car* and *Truck* mapped to the same table. This is done by mapping the ‘ids’ (e.g. *carId*, *truckId*) attributes of the subclasses as foreign keys in the *Vehicle_table* table. As mentioned previously, a primary key uniquely identifies each record (row) in a table, while a foreign key references the primary key of another table, illustrating a relationship between them. In this example the relationship represented is a one-to-one relationship, meaning that meaning that each *Vehicle* object can only be a single *Car* (identified by *carId*) or a single *Truck* (identified by *truckId*), but not both. Although, this large table stores all the data related to the *Vehicle* hierarchy in a single table, representing an inheritance relationship, which may be useful for applications with relatively small datasets. However, it wastes space, causing the data not to be inconsistent, as shown by the empty cells (e.g. in the *Truck* object rows, the cells corresponding to the column *carId* and *carType* are always NULL), potentially affecting system performance. "This problem can be exacerbated with deeper inheritance hierarchies" (Unhelkar, 2018, p. 223), because mapping a large number of classes to a single table can lead to increased traffic during Create, Read, Update, and Delete (CRUD) operations. This, in turn, can cause lock contention and object conflicts compromising the data integrity and system performance issues.

Figure 4

The Two-Table Design



carId	vinNum	carType
9876	asd1234	sedan
8765	qwe2345	coop

truckId	vinNum	truckType
7654	zxc3456	compact pickup
6543	qwe2345	full-size pickup

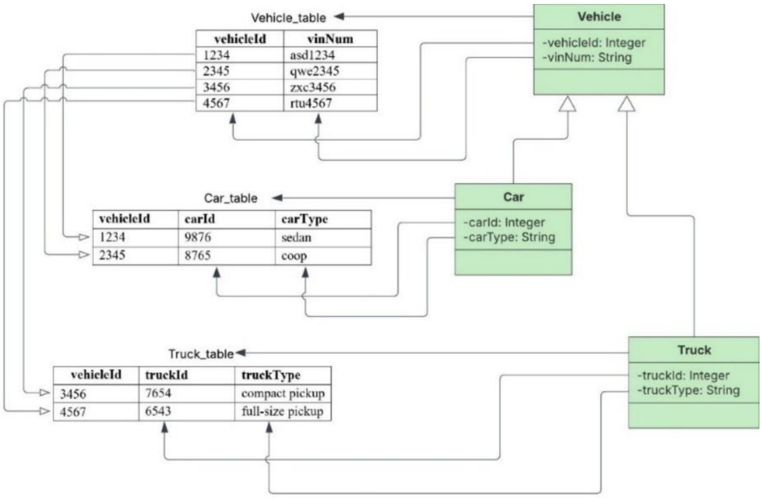
Note: The figure illustrates the mapping of the classes *Vehicle*, *Car*, and *Truck* to the two-table design. It is important to notice the design features a *Car_table table* and a *Truck_table* table but not a *Vehicule_table* table and the attribute *vinNum* from the *Vehicle* class.

Two-Table Design

An alternative to the single-table design is the two-table design illustrated in Figure 4. In this table design, the attributes from the classes *Car* and *Truck* are mapped to the columns of their respective tables. Additionally, the attribute *vinNum* from the is mapped to the *Car_table* and *Truck_table* tables. Although this solution preserves data consistency[1] and integrity[2], it does not reflect the inheritance relations that exist between the superclass *Vehicle* and its subclasses *Car* and *Truck* as their tables do not store the *Vehicle_table* data primary key values. Thus, this design is not the best design to represent the relationship between classes.

Figure 5

The Three-Table Design



Vehicle_table

vehicleId	vinNum
1234	asd1234
2345	qwe2345
3456	zxc3456
4567	rtu4567

Car_table

vehicleId	carId	carType
1234	9876	sedan
2345	8765	coop

Truck_table

vehicleId	truckId	truckType
3456	7654	compact pickup
4567	6543	full-size pickup

Note: The figure illustrates the mapping of the classes *Vehicle*, *Car,* and *Truck* to the two-table design. It is important to notice the design features a *Car_table table* and a *Truck_table* table but not a *Vehicle_table* table and the attribute *vinNum* from the *Vehicle* class.

[1] "Data consistency refers to the state of data in which all copies or instances are the same across all systems and databases. Consistency helps ensure that data is accurate, up-to-date and coherent across different database systems, applications and platforms" (Arnold, 2023).

[2] "Data integrity refers to the accuracy, completeness and consistency of data throughout its life cycle. It is the assurance that the data has not been tampered with or altered in any unauthorized way. In other words, data integrity helps ensure that the data remains intact, uncorrupted and reliable" (Arnold, 2023).

Three-Table Design

A better design option to represent relationships between classes, preserve data consistency and integrity, and maintain system performance is the three-table design, In Figure 5, all class and their attribute are mapped to their respective tables. The *Car_table* and *Truck_table* tables store the primary key values from the *Vehicle_table* table as foreign key values representing a relationship between the *Vehicle_table* table and each of the other two tables. This relationship is a one-to-one relationship simulating a “is a” relationship. In other words, through the use of the *Vehicule_table* primary key values (*VehiculeId*) as foreign key values in the *Car_table* and *Truck_table* tables, ORM simulates the inheritance relationship where the *Car* and *Truck* objects are a child of the parent *Vehicle*, in other terms, the *Car* and *Truck* classes are subclasses of the superclass *Vehicle.* In other words, the three-table design simulates inheritance in relational databases by using foreign keys making them a better choice than single-table and two-table designs for representing object hierarchies.

To summarize this step, the three table designs for mapping the *Vehicle*, *Car*, and *Truck* inheritance hierarchy to relational database tables show that the *Single-Table* design places all attributes in one table (*Vehicle_table*) and can lead to redundancy and potential inconsistencies that may affect a system perforce, especially in systems with large datasets. The Two-Table design creates separate tables for subclasses (*Car_table*, *Truck_table*). This improves data consistency from the single-table design; however, it does not fully represent the inheritance relationship because subclass tables lack the *Vehicle*'s primary key. On the other hand, the Three-Table design separates the tables creating a table for each class with subclass tables containing the superclass primary key values as foreign key values. This table indexing method reflects the model's inheritance and preserves data integrity and consistency.

Additionaly, this approach also enhances system performance as it minimizes traffic (Unhelkar, 2018). For all these reasons, the three-table design is the best choice for modeling complex object systems with inheritance relationships within a relational database.

Step-3 Mapping Multiplicity to Relational Databases

In the context of OOP, multiplicity defines the number of instances of one class that can be associated with a single instance of another class within a particular relationship (Greeff & Ghoshal, 2004). These multiplicities can be one-to-one, one-to-many, many-to-one, or many-to-many relationships. The table below provides a description and example of these relationships in the context of relational databases.

Table 5

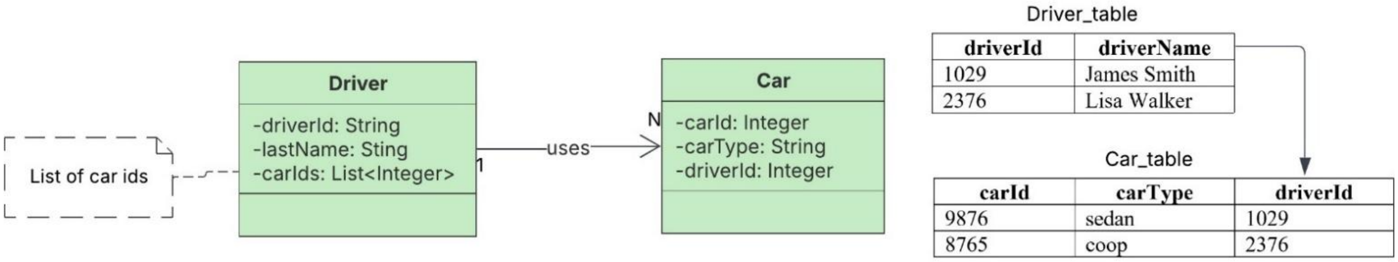
Multiplicity Relationships

Relationship Type	Description	Example
1:1 One-to-one	One record in a table is associated with one and only one record in another table.	In a government database, each person has one and only one passport, and each passport is assigned to only one person .
1:N One-to-many	One record in a table can be associated with one or more records in another table, but each record in the second table is associated with only one record in the first table.	In an e-commerce database, one customer can place many orders, but each order is associated with only one customer .
N:1 Many-to-one	Multiple records in one table can be associated with one record in another table.	Many employees can work in one department .
N:N Many-to-many	Multiple records in one table can be associated with multiple records in another table.	Students can enroll in many courses, and each course can have many students.

Note: the table provides descriptions of the different multiplicity relationships, as well as, examples. From several sources (Virgo, 2025; IBM, 2021c; Nalimov, 2024; Claris, 2024)

Figure 6

Driver-Car One-To-Many



Driver_table

driverId	driverName
1029	James Smith
2376	Lisa Walker

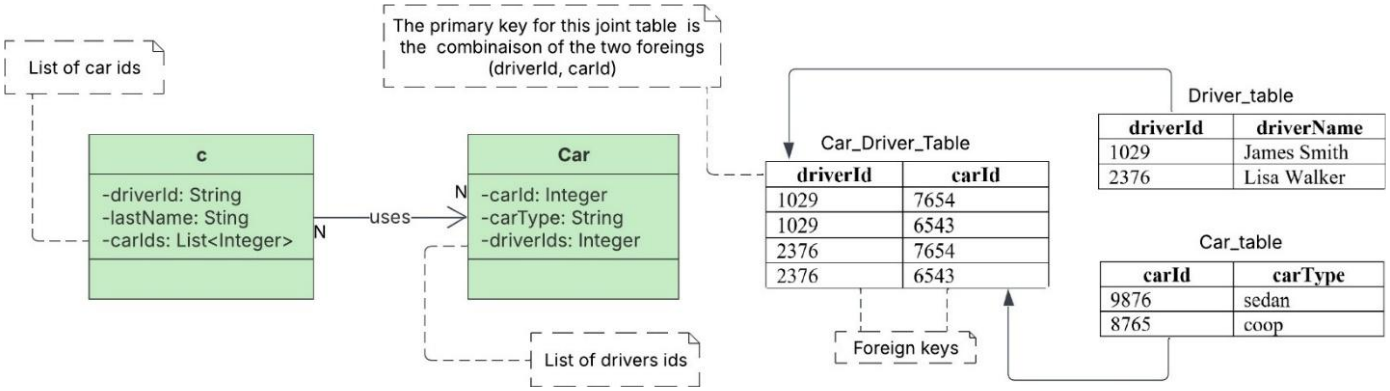
Car_table

carId	carType	driverId
9876	sedan	1029
8765	coop	2376

Note: the figure illustrates a driver-car one-to-many multiplicity relationship, where a driver can use (drive) many cars.

Figure 7

Driver-Car Many-To-Many



Car_Driver_Table

driverId	carId
1029	7654
1029	6543
2376	7654
2376	6543

Driver_table

driverId	driverName
1029	James Smith
2376	Lisa Walker

Car_table

carId	carType
9876	sedan
8765	coop

Note: The figure illustrates a driver-car many-to-many multiplicity relationship, where a driver can use (drive) many cars and a car can have many drivers.

Note that when mapping many-to-many relationships in relational databases, requires the implementation of a link table (*Car_Driver_Table*) that combines the primary keys of the associated tables as foreign keys.

Mapping One-To-Many Multiplicity

In Figure 6, the *Driver* and *Car* classes association relationship with a multiplicity of one-to-many is mapped to the tables by adding the primary key values from the *Driver_table* to the *Car_table* as foreign key values. In Figure 7, the *Driver* and *Car* classes association relationship with a multiplicity of many-to-many is mapped to the tables. This is done by creating the table *Car_Driver_Table* which links the primary key values from both tables. Note that the primary key values from the two tables are considered foreign keys and the the primary key values for the link table are a combination of the primary key values from the tables. See Code Snippet 1 for an example of how to create the table in SQL.

Code Snippet 1

SQL Code For The Car_Driver_Table

```
-- Create the link table Car_Driver_Table
CREATE TABLE Car_Driver_Table (
    driverId INT,
    carId INT,
    PRIMARY KEY (driverId, carId), -- primary key
    CONSTRAINT fk_driver FOREIGN KEY (driverId) REFERENCES Driver(driverId),
    CONSTRAINT fk_car FOREIGN KEY (carId) REFERENCES Car(carId)
);

-- Insert data into the link table
INSERT INTO Car_Driver_Table (driverId, carId) VALUES (1029, 9876);
INSERT INTO Car_Driver_Table (driverId, carId) VALUES (1029, 8765);
INSERT INTO Car_Driver_Table (driverId, carId) VALUES (2376, 9876);
INSERT INTO Car_Driver_Table (driverId, carId) VALUES (2376, 8765);
```

Note: The SQL code creates the *Car_Driver_Table* and stores the *driverId* and *carId* values to link drivers and cars together, representing the many-to-many relationship between them.

Step-4 Modify Class Diagram Driver and Car Classes CRUD Operations

Only classes and class attributes can be mapped to relational databases and class-to-class relationships can be simulated through the use of primary and foreign keys, what was not addressed yet by this essay is object behavior in the form of class methods. Object behavior can be simulated within the context of a relation database using CRUD operations. CRUD stands for

- Create—creates an object.
- Read—search for an object (record) from storage based on a criterion (key).

- Update—search and update objects (records).
- Delete—locates and remove a persistent object

(Unhelkar, 2018; Ricciardi, 2025)

These operations are executed using query languages such as Structured Query Language (SQL). Thus, it is important when designing a software system that uses databases to separate application-specific behaviors from database-specific behaviors. ULM allows the separation of persistent data operations from a system or business logic using stereotypes such as <<*entity*>> for classes that deal with the behavior of the system and <<*table*>> that deal with relational database table CRUD operation. It is also best practice to use control classes that will act as intermediaries between <<*entity*>> and <<*table*>> classes. These classes provide an interface that effectively routes CRUD operations. UML uses the stereotype <<control>> to label these intermediary classes, which are often referred to as managers or controllers. Implementing control classes "ensures that the application will remain totally separate from the database, offering advantages in terms of flexibility and reducing the impact of changes in the application on the database, as well as changes in the database on the application" (Unhelkar, 2018, p. 222). However, they also add performance overhead as they need to be executed at run time.

To summarize this step, object behavior (methods) in a relational database is represented using CRUD operations (Create, Read, Update, Delete) executed via SQL. UML stereotypes (<<entity>>, <table>>, <<control>>) are used to separate business logic from database interactions. Additionally, <<control>> classes (managers/controllers) act as intermediaries, improving flexibility but adding runtime overhead.

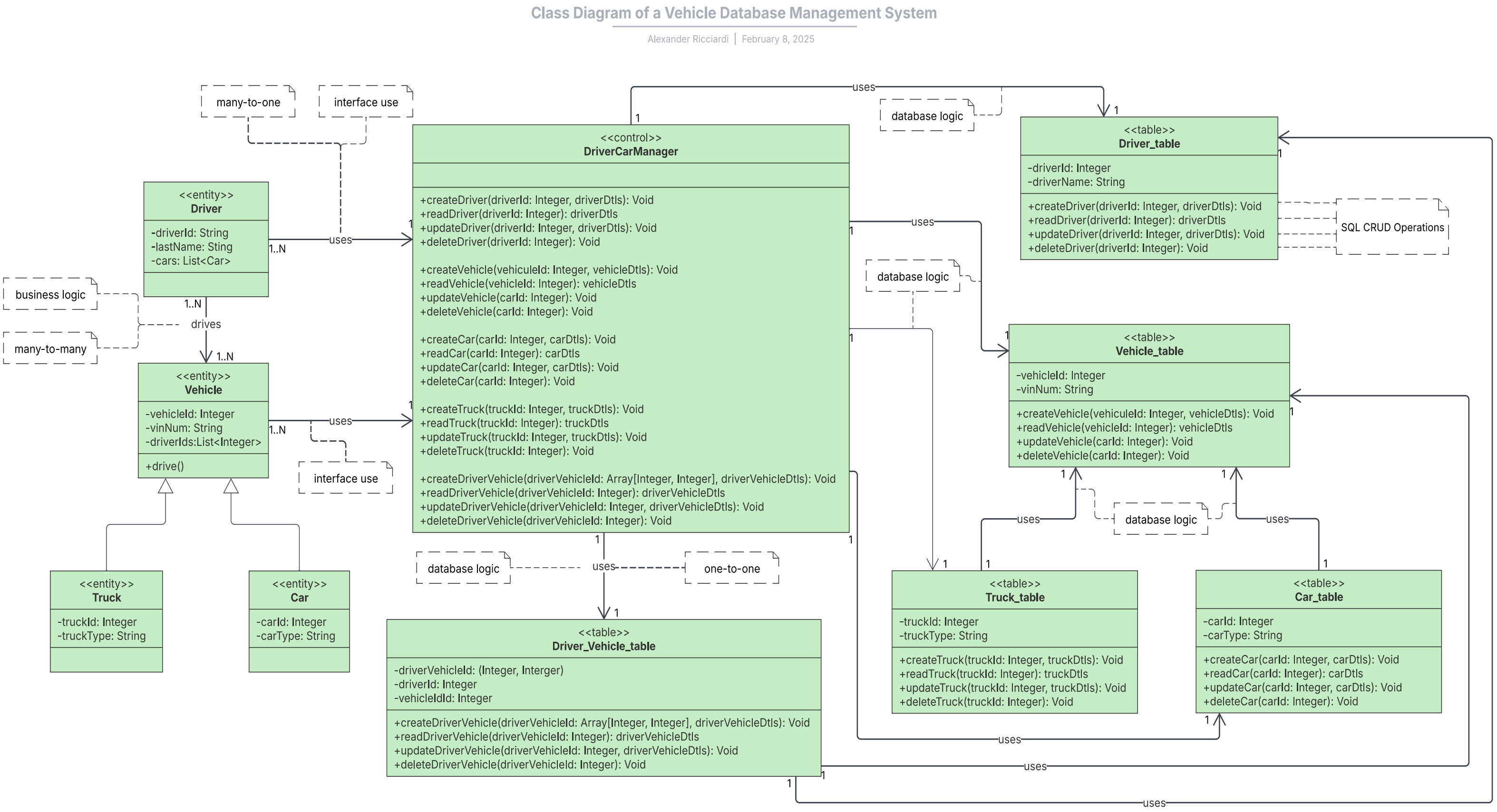
To summarize this step, the process of mapping multiplicity from OO concepts to relational databases is based on the one-to-one, one-to-many, many-to-one, and many-to-many relationships. The one-to-many relationship is mapped by adding a foreign key to the table representing the "many" side and a many-to-many relationship is mapped using a link table with a composite of the primary keys from the linked tables. The SQL code snippet provides an example of how a link table (*Car_Driver_Table*) for a many-to-many relationship can be handled using SQL code.

Step-5 Final Class Diagram and Tables

In this section, all the concepts discussed previously are implemented within an example of a class diagram of a vehicle database management system and a set of tables.

Figure 8

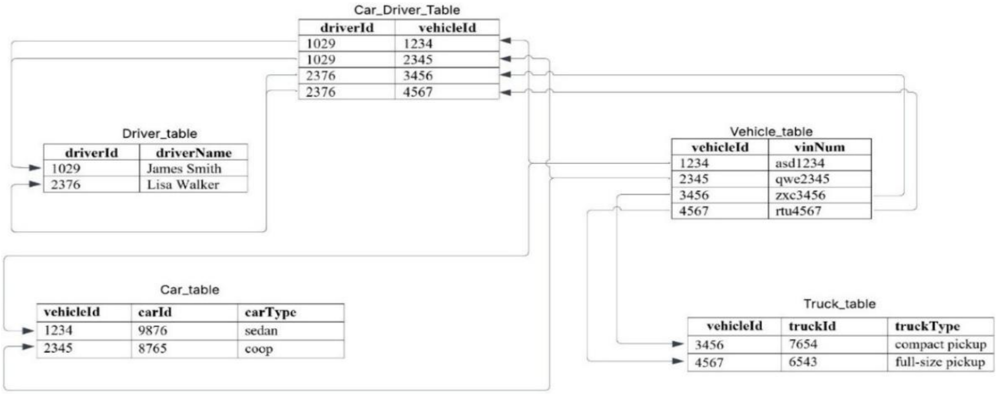
Vehicle Database Management System Class Diagram


Class Diagram of a Vehicle Database Management System
Alexander Ricciardi | February 8, 2025
many-to-one
interface use
<<entity>>
Driver
-driverId: String
-lastName: Sting
-cars: List<Car>
uses
1..N
1
business logic
drives
many-to-many
1..N
<<entity>>
Vehicle
-vehicleId: Integer
-vinNum: String
-driverIds:List<Integer>
+drive()
uses
1..N
interface use
<<entity>>
Truck
-truckId: Integer
-truckType: String
<<entity>>
Car
-carId: Integer
-carType: String
<<control>>
DriverCarManager
+createDriver(driverId: Integer, driverDtls): Void
+readDriver(driverId: Integer): driverDtls
+updateDriver(driverId: Integer, driverDtls): Void
+deleteDriver(driverId: Integer): Void
+createVehicle(vehiculeId: Integer, vehicleDtls): Void
+readVehicle(vehicleId: Integer): vehicleDtls
+updateVehicle(carId: Integer): Void
+deleteVehicle(carId: Integer): Void
+createCar(carId: Integer, carDtls): Void
+readCar(carId: Integer): carDtls
+updateCar(carId: Integer, carDtls): Void
+deleteCar(carId: Integer): Void
+createTruck(truckId: Integer, truckDtls): Void
+readTruck(truckId: Integer): truckDtls
+updateTruck(truckId: Integer, truckDtls): Void
+deleteTruck(truckId: Integer): Void
+createDriverVehicle(driverVehicleId: Array[Integer, Integer], driverVehicleDtls): Void
+readDriverVehicle(driverVehicleId: Integer): driverVehicleDtls
+updateDriverVehicle(driverVehicleId: Integer, driverVehicleDtls): Void
+deleteDriverVehicle(driverVehicleId: Integer): Void
database logic
uses
one-to-one
1
<<table>>
Driver_Vehicle_table
-driverVehicleId: (Integer, Interger)
-driverId: Integer
-vehicleIdId: Integer
+createDriverVehicle(driverVehicleId: Array[Integer, Integer], driverVehicleDtls): Void
+readDriverVehicle(driverVehicleId: Integer): driverVehicleDtls
+updateDriverVehicle(driverVehicleId: Integer, driverVehicleDtls): Void
+deleteDriverVehicle(driverVehicleId: Integer): Void
uses
database logic
<<table>>
Driver_table
-driverId: Integer
-driverName: String
+createDriver(driverId: Integer, driverDtls): Void
+readDriver(driverId: Integer): driverDtls
+updateDriver(driverId: Integer, driverDtls): Void
+deleteDriver(driverId: Integer): Void
SQL CRUD Operations
uses
database logic
<<table>>
Vehicle_table
-vehicleId: Integer
-vinNum: String
+createVehicle(vehiculeId: Integer, vehicleDtls): Void
+readVehicle(vehicleId: Integer): vehicleDtls
+updateVehicle(carId: Integer): Void
+deleteVehicle(carId: Integer): Void
uses
database logic
uses
<<table>>
Truck_table
-truckId: Integer
-truckType: String
+createTruck(truckId: Integer, truckDtls): Void
+readTruck(truckId: Integer): truckDtls
+updateTruck(truckId: Integer, truckDtls): Void
+deleteTruck(truckId: Integer): Void
<<table>>
Car_table
-carId: Integer
-carType: String
+createCar(carId: Integer, carDtls): Void
+readCar(carId: Integer): carDtls
+updateCar(carId: Integer, carDtls): Void
+deleteCar(carId: Integer): Void
uses
uses
uses

Note: The figure illustrates the final class diagram of the vehicle database management system.

Figure 9

Vehicle Database Management System Tables



Car_Driver_Table

driverId	vehicleId
1029	1234
1029	2345
2376	3456
2376	4567

Driver_table

driverId	driverName
1029	James Smith
2376	Lisa Walker

Vehicle_table

vehicleId	vinNum
1234	asd1234
2345	qwe2345
3456	zxc3456
4567	rtu4567

Car_table

vehicleId	carId	carType
1234	9876	sedan
2345	8765	coop

Truck_table

vehicleId	truckId	truckType
3456	7654	compact pickup
4567	6543	full-size pickup

Note: The figure illustrates the set of tables used by the vehicle database management system.

Figure 8 illustrates the final UML class diagram of the vehicle database management system. It incorporates the classes *Vehicle*, *Car*, *Truck*, and *Driver* which are represented with the <<*entity*>> stereotype. The relationships between these classes include both inheritance and many-to-many associations. These classes implement the business logic of the system. The database logic is implemented by the *Car_Driver_Table*, *Driver_table*, *Vehicle_table*, *Car_table*, and *Truck_table* which are represented with the <<*table*>> stereotype. Their role is to simulate the relationships and behaviors of the *Car* and *Driver* classes. Additionally, the *VehicleManager* class represented with the <<*control*>> stereotype acts as an abstraction between the system business logic and the system database logic by providing an interface that encapsulates and manages the CRUD operations.

Figure 9 shows the final set of tables for the vehicle database management system. This design utilizes the three-table approach for inheritance, with separate tables for *Vehicle*, *Car*, and *Truck* objects. The many-to-many relationship between *Driver* and *Car* objects is implemented using the *Car_Driver_Table* table, which stores the associations between specific drivers and cars by storing the primary key values from both the *Driver_table* and *Vehicle_table* tables. The inheritance relationships between the superclass *Vehicle* and its subclasses *Car* and *truck* is simulated by mapping the *Vehicle_table* primary key values to both *Driver_table* and *Vehicle_table* tables as foreign key values.

To summarize this step, when implementing a relational database it is important to separate business or system logic (entity classes) from database logic (table classes) using UML stereotypes such as <<*entity*>>, <<*control*>>, and <*table*>>. Control classes act as intermediaries between the classes modeling the business logic and the ones modeling the database logic. They manage CRUD operations improving the flexibility of the system.

Conclusion

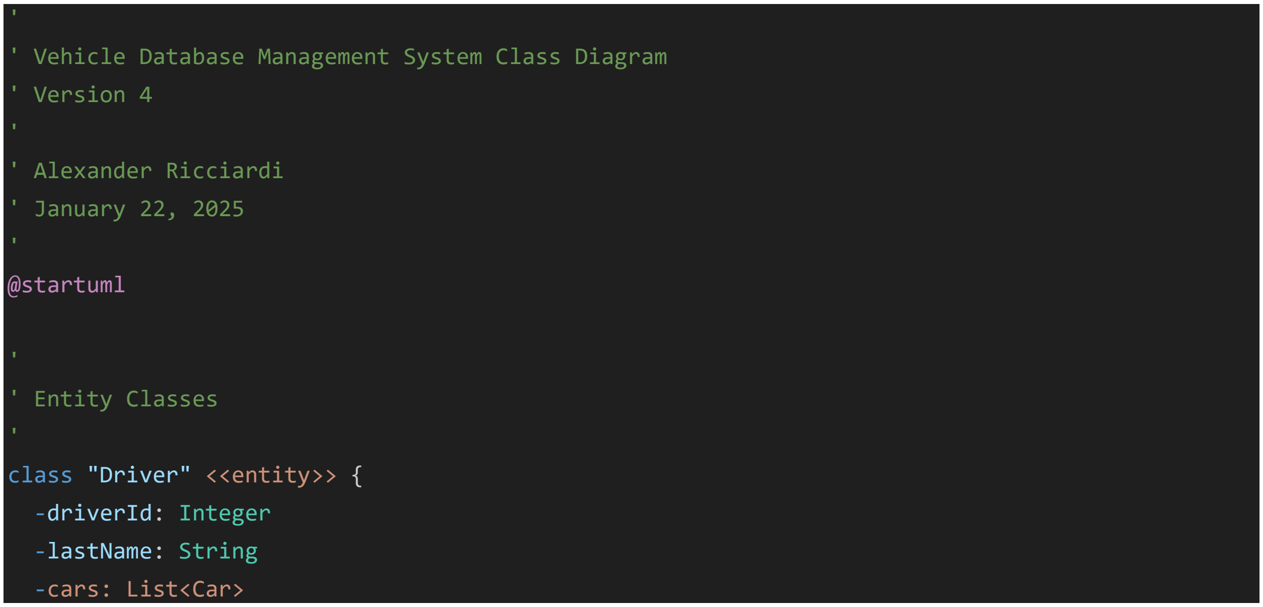
Databases allow data to be stored persistently and structured in a way that can be retrieved reliably for use, manipulation, and analysis. OODBMS and NoSQL databases offer advantages for specific data types. However, Relational Database Management Systems (RDBMS) remain prevalent due to their reliability, standardization, and the wide usage of SQL. Using UML class diagrams, ORM techniques, and a vehicle database management system example based on the RDBMS model, this essay explored various inheritance mapping strategies (single-table, two-table, and three-table) and the representation of one-to-many and many-to-many relationships. It also highlighted the importance of separating business logic from database logic using UML control, entity, and table class concepts and stereotypes. Ultimately, the choice of using relational databases with a three-table design to model the vehicle database management system example is based on this approach's ability to represent object inheritance, ensure data integrity, and provide query performance, while preserving data consistency and integrity. This highlights how important it is to understand how to map object-oriented concepts to relational database schemas for building modern applications.

Part-2 Generated Pseudocode

This section showcases the pseudocode of a vehicle database management system based on the final class diagram from the essay in part-1. The pseudocode was translated from PlantUML code using a small Java program named: “PlantUML Class Diagram To Pseudocode Generator Version-5”. The program translates PlantUML class diagrams to pseudocode by parsing the PlantUML code using Regex and Java methods. It also generates comments based on CRUD operations. PlanULM is a simple and intuitive language that is used to drafts of various types of diagrams, the language supports UML diagrams (PlantUML, n.d.). This part of the assignment was the most challenging. To translate the code from PlantUML to pseudocode the PlantUML code lines needed to be first parsed in terms (strings with meaning) that can be translated. For this purpose, I mostly used Regular Expressions (Regex) as it was the most efficient method. To help me formulate Regex, I used a tool called Regex101 (n.d.). Bellow the PlantUML code snippet and the output from the code contained in the code snippet.

Code Snippet 2

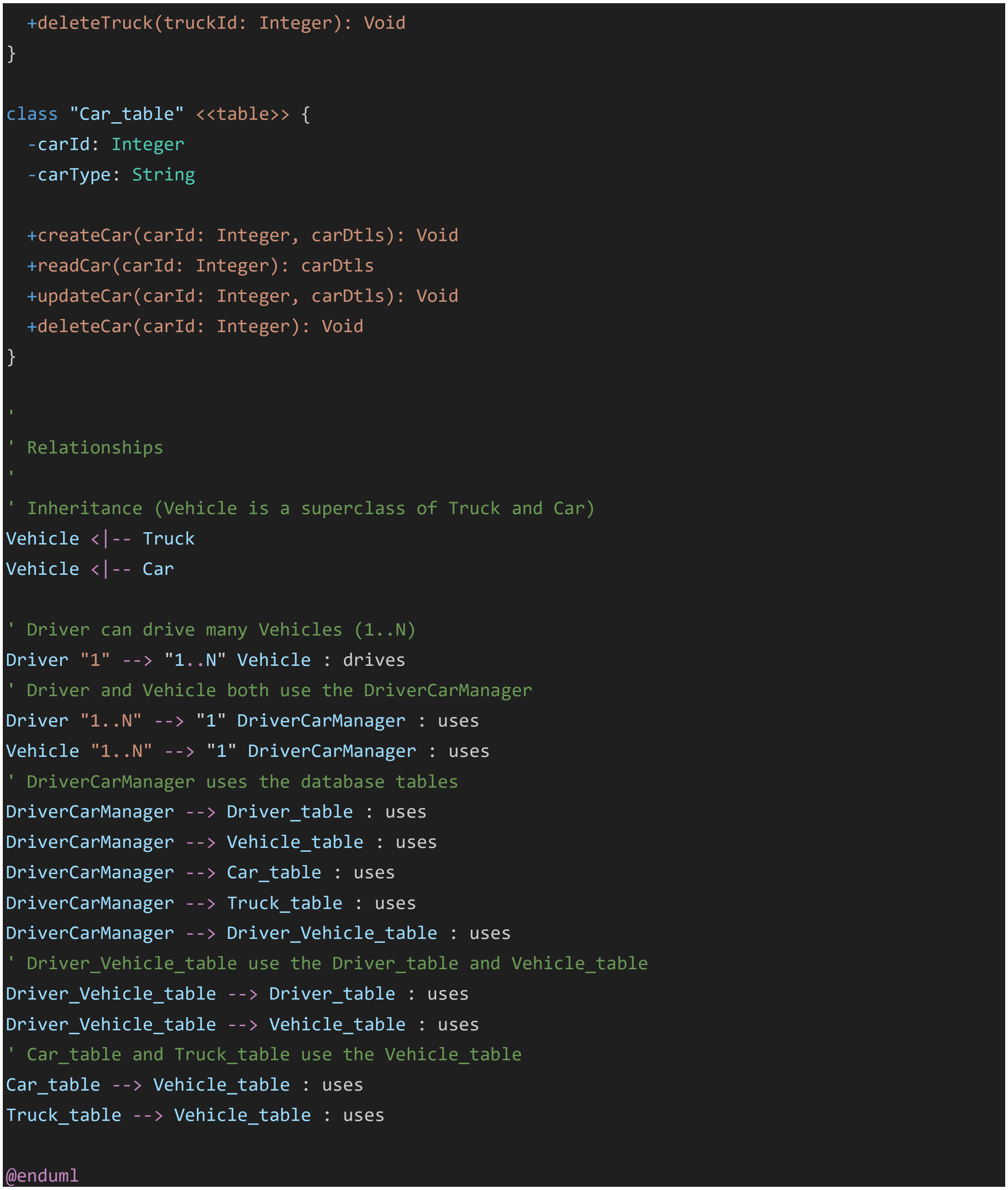
PlantUML Code For The Vehicle Database Management System Class Diagram



```
'
' Vehicle Database Management System Class Diagram
' Version 4
'
' Alexander Ricciardi
' January 22, 2025
'
@startuml

'
' Entity Classes
'
class "Driver" <<entity>> {
  -driverId: Integer
  -lastName: String
  -cars: List<Car>
```

```
}

class "Vehicle" <<entity>> {
  -vehicleId: Integer
  -vinNum: String
  -driverIds: List<Integer>
  +drive()
}

class "Truck" <<entity>> {
  -truckId: Integer
  -truckType: String
}

class "Car" <<entity>> {
  -carId: Integer
  -carType: String
}

'
' Control Class
'
class "DriverCarManager" <<control>> {
+createDriver(driverId: Integer, driverDtls): Void
+readDriver(driverId: Integer): driverDtls
+updateDriver(driverId: Integer, driverDtls): Void
+deleteDriver(driverId: Integer): Void

+createVehicle(vehiculeId: Integer, vehicleDtls): Void
+readVehicle(vehicleId: Integer): vehicleDtls
+updateVehicle(carId: Integer): Void
+deleteVehicle(carId: Integer): Void

+createCar(carId: Integer, carDtls): Void
+readCar(carId: Integer): carDtls
+updateCar(carId: Integer, carDtls): Void
+deleteCar(carId: Integer): Void

+createTruck(truckId: Integer, truckDtls): Void
+readTruck(truckId: Integer): truckDtls
+updateTruck(truckId: Integer, truckDtls): Void
+deleteTruck(truckId: Integer): Void

+createDriverVehicle(driverVehicleId: Array[Integer, Integer], driverVehicleDtls): Void
+readDriverVehicle(driverVehicleId: Integer): driverVehicleDtls
```

```
+updateDriverVehicle(driverVehicleId: Integer, driverVehicleDtls): Void
+deleteDriverVehicle(driverVehicleId: Integer): Void
}

'
' Table Classes
'
class "Driver_table" <<table>> {
  -driverId: Integer
  -driverName: String

  +createDriver(driverId: Integer, driverDtls): Void
  +readDriver(driverId: Integer): driverDtls
  +updateDriver(driverId: Integer, driverDtls): Void
  +deleteDriver(driverId: Integer): Void
}

class "Vehicle_table" <<table>> {
  -vehicleId: Integer
  -vinNum: String

  +createVehicle(vehicleId: Integer, vehicleDtls): Void
  +readVehicle(vehicleId: Integer): vehicleDtls
  +updateVehicle(carId: Integer): Void
  +deleteVehicle(carId: Integer): Void
}

class "Driver_Vehicle_table" <<table>> {
  -driverVehicleId: [Integer, Integer]
  -driverId: Integer
  -vehicleId: Integer

  +createDriverVehicle(driverVehicleId: Array[Integer, Integer], driverVehicleDtls): Void
  +readDriverVehicle(driverVehicleId: Integer): driverVehicleDtls
  +updateDriverVehicle(driverVehicleId: Integer, driverVehicleDtls): Void
  +deleteDriverVehicle(driverVehicleId: Integer): Void
}

class "Truck_table" <<table>> {
  -truckId: Integer
  -truckType: String

  +createTruck(truckId: Integer, truckDtls): Void
  +readTruck(truckId: Integer): truckDtls
  +updateTruck(truckId: Integer, truckDtls): Void
```

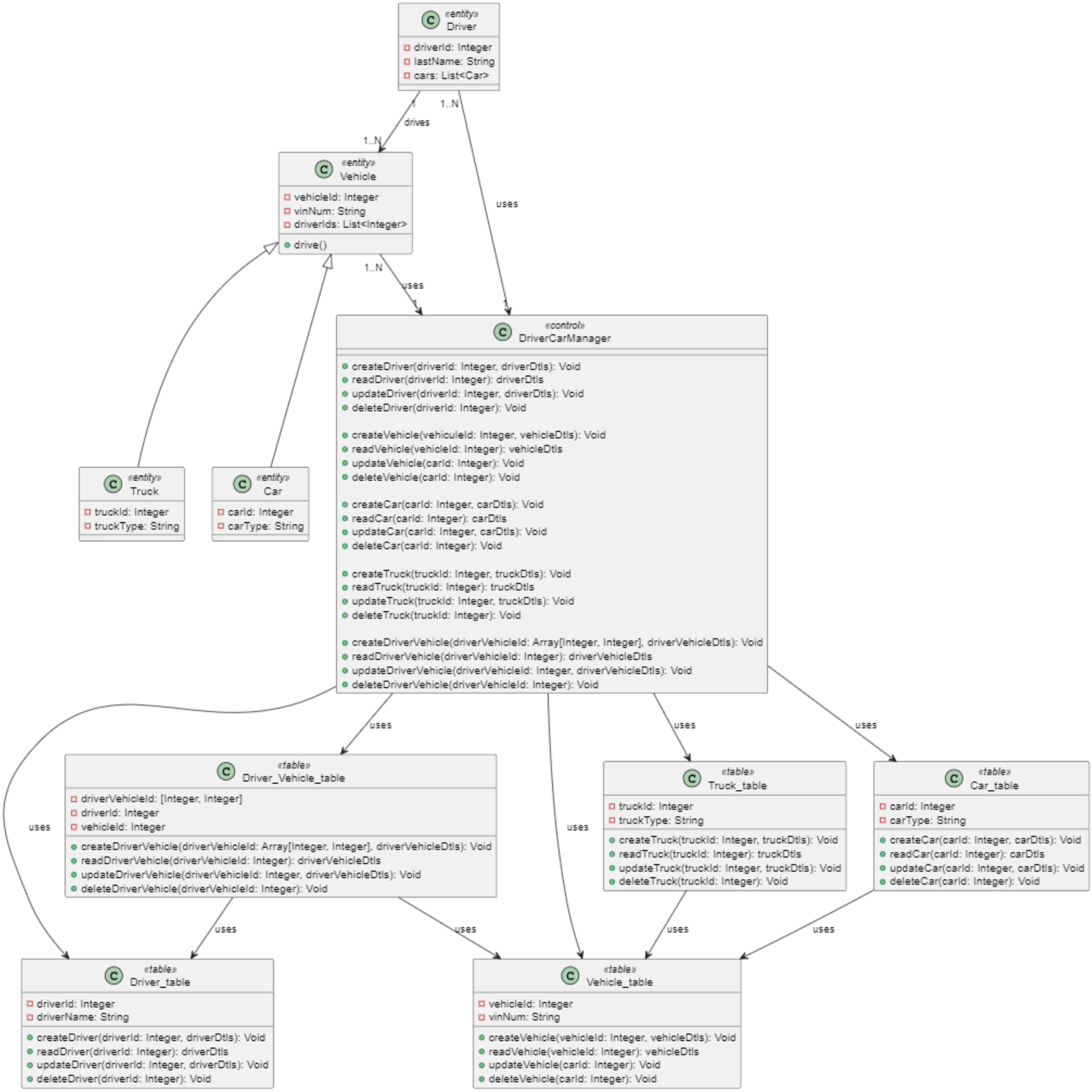


```
  +deleteTruck(truckId: Integer): Void
}

class "Car_table" <<table>> {
  -carId: Integer
  -carType: String

  +createCar(carId: Integer, carDtls): Void
  +readCar(carId: Integer): carDtls
  +updateCar(carId: Integer, carDtls): Void
  +deleteCar(carId: Integer): Void
}

'
' Relationships
'
' Inheritance (Vehicle is a superclass of Truck and Car)
Vehicle <|-- Truck
Vehicle <|-- Car

' Driver can drive many Vehicles (1..N)
Driver "1" --> "1..N" Vehicle : drives
' Driver and Vehicle both use the DriverCarManager
Driver "1..N" --> "1" DriverCarManager : uses
Vehicle "1..N" --> "1" DriverCarManager : uses
' DriverCarManager uses the database tables
DriverCarManager --> Driver_table : uses
DriverCarManager --> Vehicle_table : uses
DriverCarManager --> Car_table : uses
DriverCarManager --> Truck_table : uses
DriverCarManager --> Driver_Vehicle_table : uses
' Driver_Vehicle_table use the Driver_table and Vehicle_table
Driver_Vehicle_table --> Driver_table : uses
Driver_Vehicle_table --> Vehicle_table : uses
' Car_table and Truck_table use the Vehicle_table
Car_table --> Vehicle_table : uses
Truck_table --> Vehicle_table : uses

@enduml
```

Note: The code snippet contains the code lines outputting the vehicle database management system class diagram.

Figure 10

Class Diagram Output From PlantUML Code

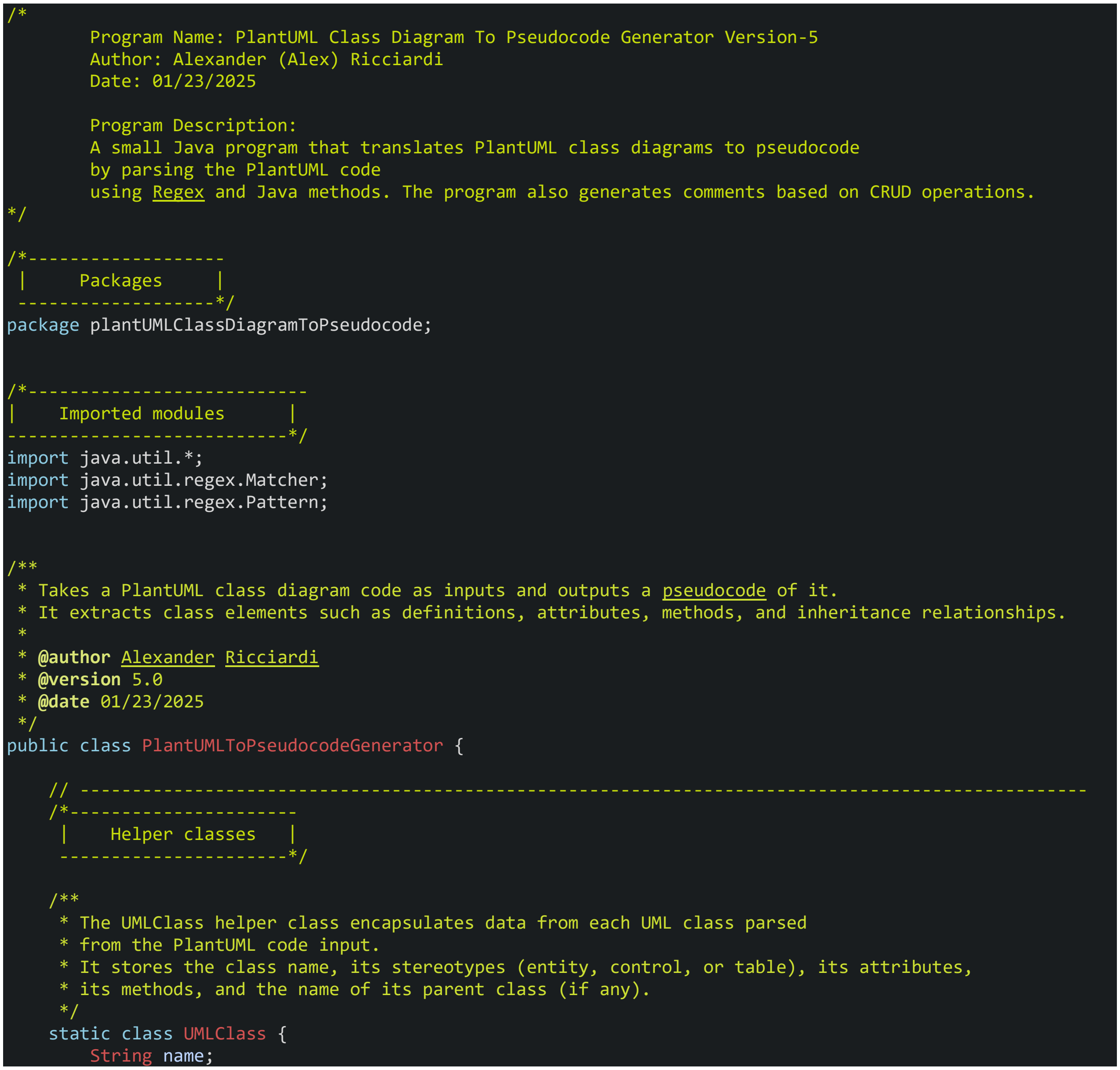

«entity» Driver
driverId: Integer
lastName: String
cars: List<Car>
1
drives
1..N
1..N
uses
«entity» Vehicle
vehicleId: Integer
vinNum: String
driverIds: List<Integer>
drive()
1..N
uses
1
1
«control» DriverCarManager
createDriver(driverId: Integer, driverDtls): Void
readDriver(driverId: Integer): driverDtls
updateDriver(driverId: Integer, driverDtls): Void
deleteDriver(driverId: Integer): Void
createVehicle(vehiculeId: Integer, vehicleDtls): Void
readVehicle(vehicleId: Integer): vehicleDtls
updateVehicle(carId: Integer): Void
deleteVehicle(carId: Integer): Void
createCar(carId: Integer, carDtls): Void
readCar(carId: Integer): carDtls
updateCar(carId: Integer, carDtls): Void
deleteCar(carId: Integer): Void
createTruck(truckId: Integer, truckDtls): Void
readTruck(truckId: Integer): truckDtls
updateTruck(truckId: Integer, truckDtls): Void
deleteTruck(truckId: Integer): Void
createDriverVehicle(driverVehicleId: Array[Integer, Integer], driverVehicleDtls): Void
readDriverVehicle(driverVehicleId: Integer): driverVehicleDtls
updateDriverVehicle(driverVehicleId: Integer, driverVehicleDtls): Void
deleteDriverVehicle(driverVehicleId: Integer): Void
«entity» Truck
truckId: Integer
truckType: String
«entity» Car
carId: Integer
carType: String
uses
uses
uses
uses
uses
«table» Driver_Vehicle_table
driverVehicleId: [Integer, Integer]
driverId: Integer
vehicleId: Integer
createDriverVehicle(driverVehicleId: Array[Integer, Integer], driverVehicleDtls): Void
readDriverVehicle(driverVehicleId: Integer): driverVehicleDtls
updateDriverVehicle(driverVehicleId: Integer, driverVehicleDtls): Void
deleteDriverVehicle(driverVehicleId: Integer): Void
«table» Truck_table
truckId: Integer
truckType: String
createTruck(truckId: Integer, truckDtls): Void
readTruck(truckId: Integer): truckDtls
updateTruck(truckId: Integer, truckDtls): Void
deleteTruck(truckId: Integer): Void
«table» Car_table
carId: Integer
carType: String
createCar(carId: Integer, carDtls): Void
readCar(carId: Integer): carDtls
updateCar(carId: Integer, carDtls): Void
deleteCar(carId: Integer): Void
uses
uses
uses
uses
«table» Driver_table
driverId: Integer
driverName: String
createDriver(driverId: Integer, driverDtls): Void
readDriver(driverId: Integer): driverDtls
updateDriver(driverId: Integer, driverDtls): Void
deleteDriver(driverId: Integer): Void
«table» Vehicle_table
vehicleId: Integer
vinNum: String
createVehicle(vehicleId: Integer, vehicleDtls): Void
readVehicle(vehicleId: Integer): vehicleDtls
updateVehicle(carId: Integer): Void
deleteVehicle(carId: Integer): Void

Note: The figure illustrates the class diagram outputted by the vehicle database management system class diagram PlanUML code.

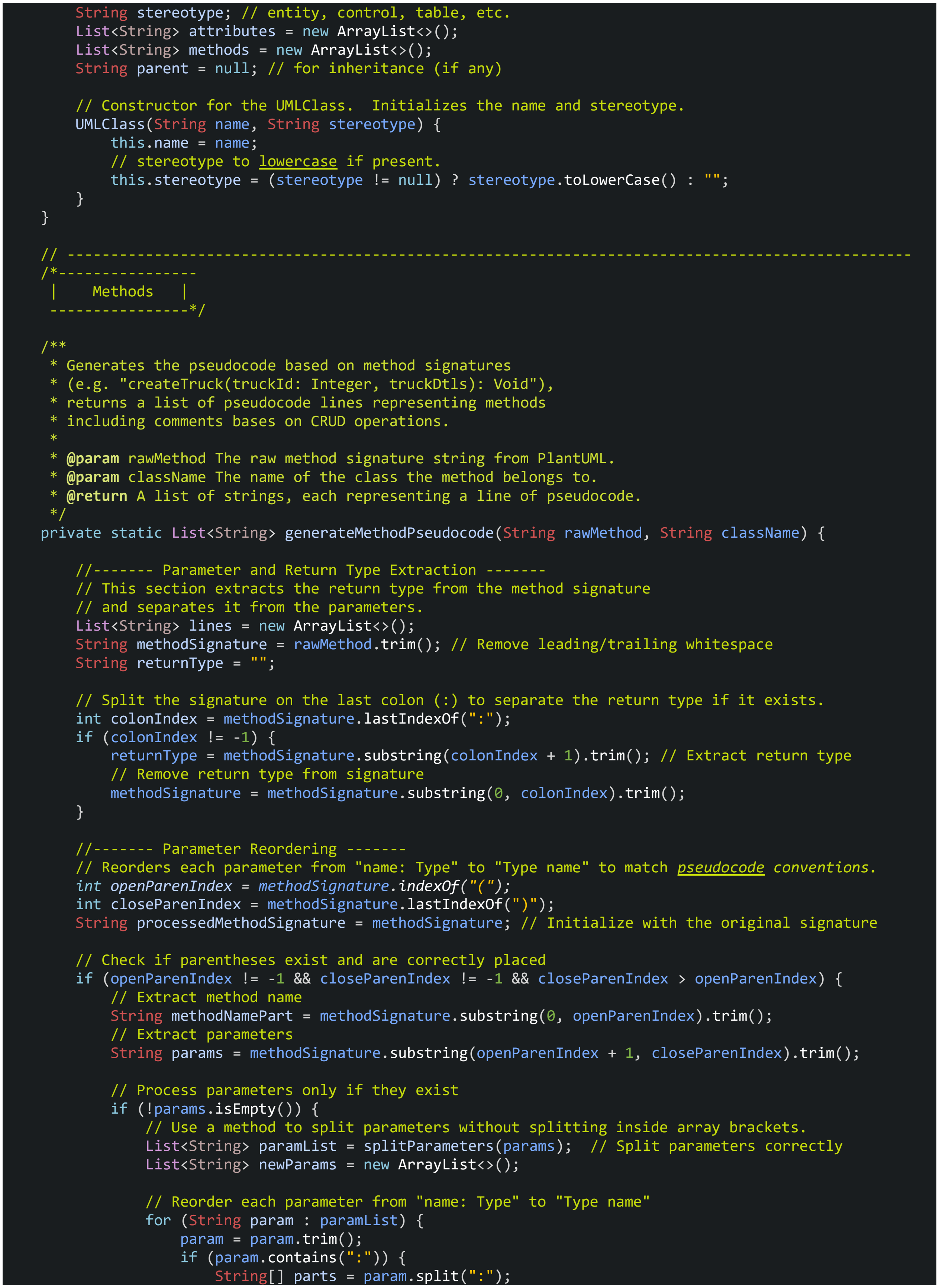
The vehicle database management system class diagram PlanUML code is integrated into the PlantUML Class Diagram To Pseudocode Generator Version-5 Java program. The program will translate the PlanUML into pseudocode and print it out on the console. Below is the code snippet of the Java program, an image of the console output of the program, and a code snippet of the outputted pseudocode:

Code Snippet 3

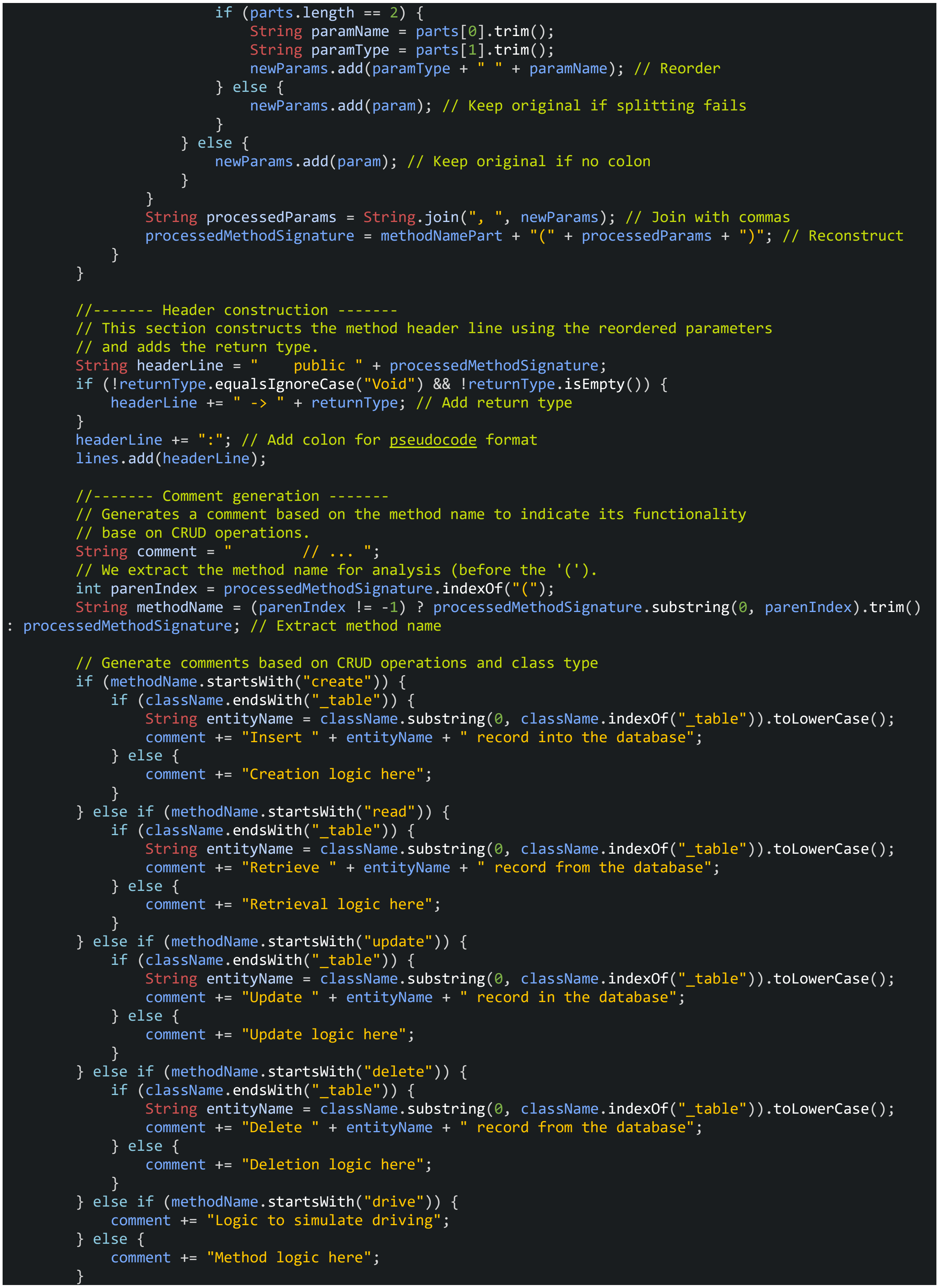
PlantUML Class Diagram To Pseudocode Generator - Java



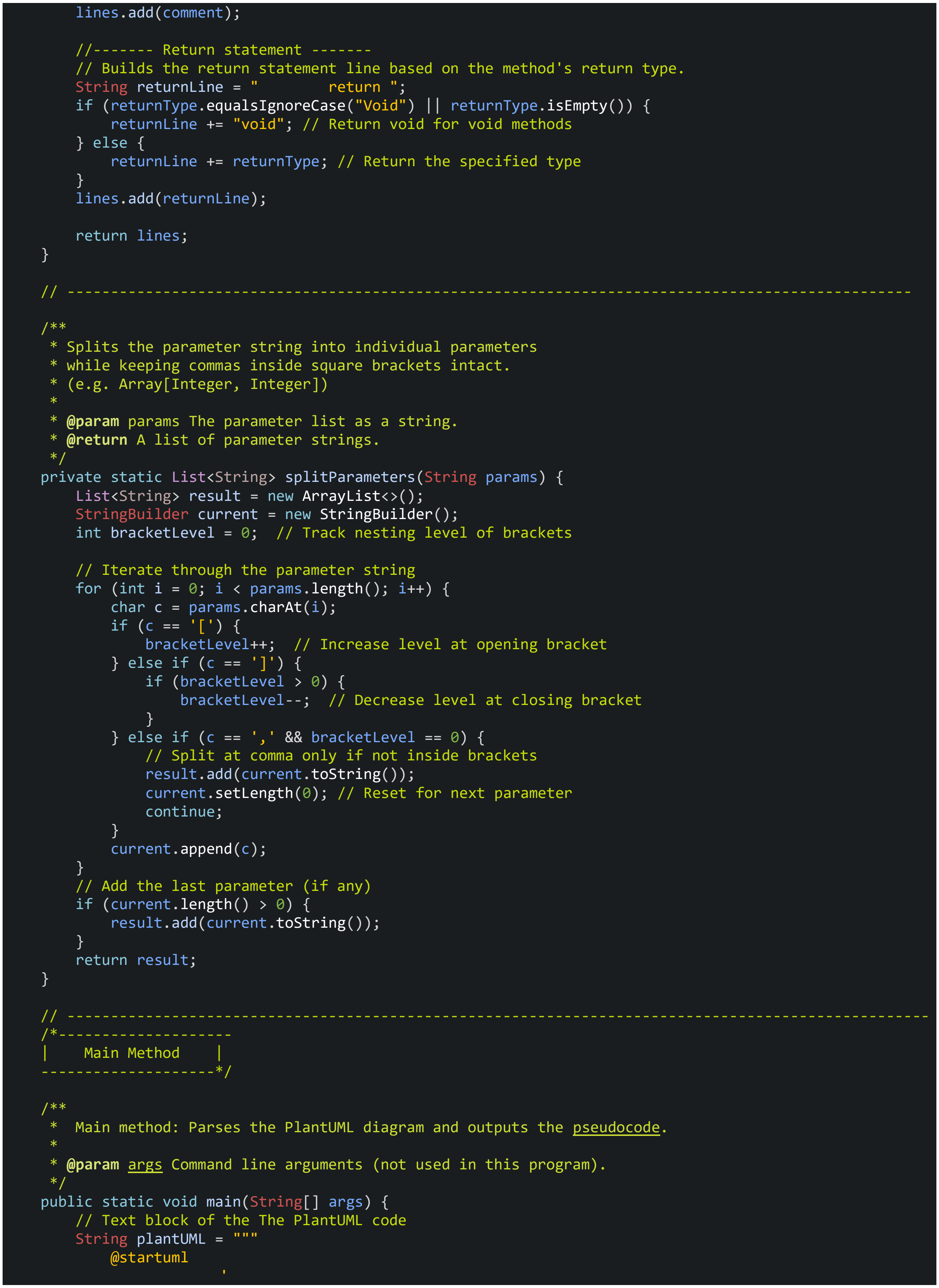
```
/*
        Program Name: PlantUML Class Diagram To Pseudocode Generator Version-5
        Author: Alexander (Alex) Ricciardi
        Date: 01/23/2025

        Program Description:
        A small Java program that translates PlantUML class diagrams to pseudocode
        by parsing the PlantUML code
        using Regex and Java methods. The program also generates comments based on CRUD operations.
*/

/*-------------------
 |     Packages     |
 -------------------*/
package plantUMLClassDiagramToPseudocode;


/*---------------------------
|    Imported modules      |
---------------------------*/
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;


/**
 * Takes a PlantUML class diagram code as inputs and outputs a pseudocode of it.
 * It extracts class elements such as definitions, attributes, methods, and inheritance relationships.
 *
 * @author Alexander Ricciardi
 * @version 5.0
 * @date 01/23/2025
 */
public class PlantUMLToPseudocodeGenerator {

    // -------------------------------------------------------------------------------------------------
    /*----------------------
     |    Helper classes   |
     ----------------------*/

    /**
     * The UMLClass helper class encapsulates data from each UML class parsed
     * from the PlantUML code input.
     * It stores the class name, its stereotypes (entity, control, or table), its attributes,
     * its methods, and the name of its parent class (if any).
     */
    static class UMLClass {
        String name;
```



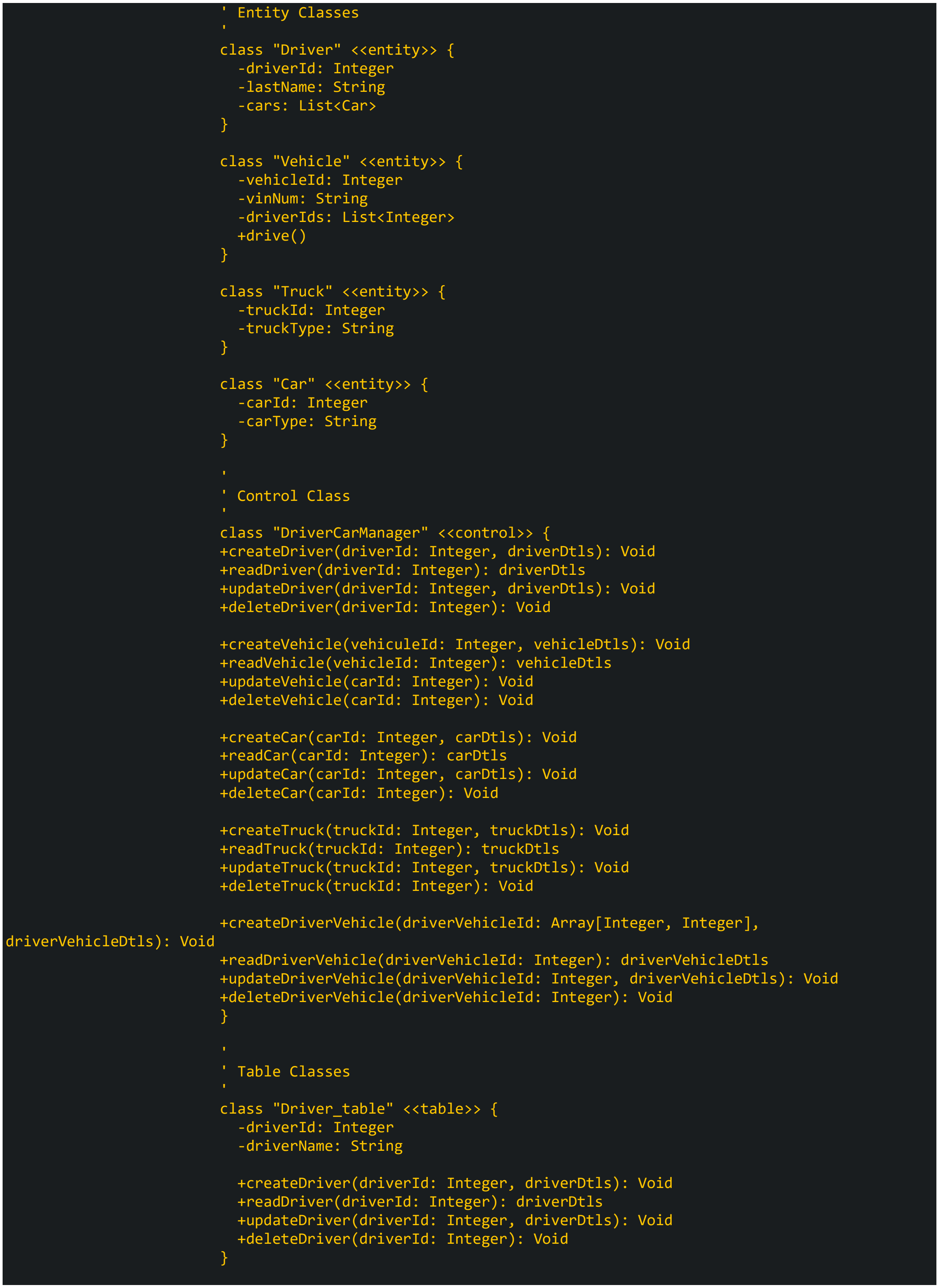
```
        String stereotype; // entity, control, table, etc.
        List<String> attributes = new ArrayList<>();
        List<String> methods = new ArrayList<>();
        String parent = null; // for inheritance (if any)

        // Constructor for the UMLClass.  Initializes the name and stereotype.
        UMLClass(String name, String stereotype) {
            this.name = name;
            // stereotype to lowercase if present.
            this.stereotype = (stereotype != null) ? stereotype.toLowerCase() : "";
        }
    }

    // -------------------------------------------------------------------------------------------------
    /*----------------
     |    Methods    |
     ----------------*/

    /**
     * Generates the pseudocode based on method signatures
     * (e.g. "createTruck(truckId: Integer, truckDtls): Void"),
     * returns a list of pseudocode lines representing methods
     * including comments bases on CRUD operations.
     *
     * @param rawMethod The raw method signature string from PlantUML.
     * @param className The name of the class the method belongs to.
     * @return A list of strings, each representing a line of pseudocode.
     */
    private static List<String> generateMethodPseudocode(String rawMethod, String className) {

        //------- Parameter and Return Type Extraction -------
        // This section extracts the return type from the method signature
        // and separates it from the parameters.
        List<String> lines = new ArrayList<>();
        String methodSignature = rawMethod.trim(); // Remove leading/trailing whitespace
        String returnType = "";

        // Split the signature on the last colon (:) to separate the return type if it exists.
        int colonIndex = methodSignature.lastIndexOf(":");
        if (colonIndex != -1) {
            returnType = methodSignature.substring(colonIndex + 1).trim(); // Extract return type
            // Remove return type from signature
            methodSignature = methodSignature.substring(0, colonIndex).trim();
        }

        //------- Parameter Reordering -------
        // Reorders each parameter from "name: Type" to "Type name" to match pseudocode conventions.
        int openParenIndex = methodSignature.indexOf("(");
        int closeParenIndex = methodSignature.lastIndexOf(")");
        String processedMethodSignature = methodSignature; // Initialize with the original signature

        // Check if parentheses exist and are correctly placed
        if (openParenIndex != -1 && closeParenIndex != -1 && closeParenIndex > openParenIndex) {
            // Extract method name
            String methodNamePart = methodSignature.substring(0, openParenIndex).trim();
            // Extract parameters
            String params = methodSignature.substring(openParenIndex + 1, closeParenIndex).trim();

            // Process parameters only if they exist
            if (!params.isEmpty()) {
                // Use a method to split parameters without splitting inside array brackets.
                List<String> paramList = splitParameters(params);  // Split parameters correctly
                List<String> newParams = new ArrayList<>();

                // Reorder each parameter from "name: Type" to "Type name"
                for (String param : paramList) {
                    param = param.trim();
                    if (param.contains(":")) {
                        String[] parts = param.split(":");
```



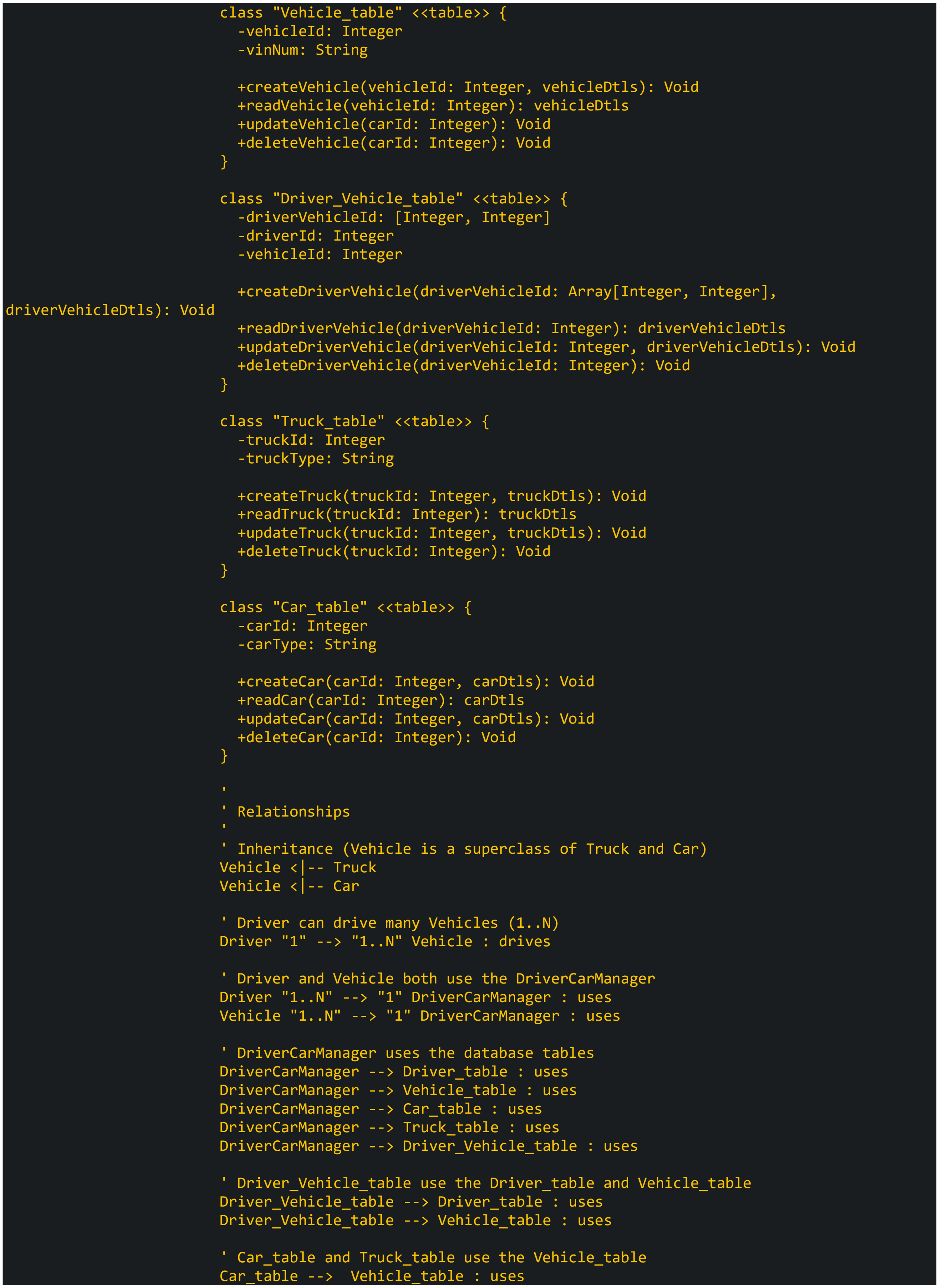
```
                        if (parts.length == 2) {
                            String paramName = parts[0].trim();
                            String paramType = parts[1].trim();
                            newParams.add(paramType + " " + paramName); // Reorder
                        } else {
                            newParams.add(param); // Keep original if splitting fails
                        }
                    } else {
                        newParams.add(param); // Keep original if no colon
                    }
                }
                String processedParams = String.join(", ", newParams); // Join with commas
                processedMethodSignature = methodNamePart + "(" + processedParams + ")"; // Reconstruct
            }
        }

        //------- Header construction -------
        // This section constructs the method header line using the reordered parameters
        // and adds the return type.
        String headerLine = "    public " + processedMethodSignature;
        if (!returnType.equalsIgnoreCase("Void") && !returnType.isEmpty()) {
            headerLine += " -> " + returnType; // Add return type
        }
        headerLine += ":"; // Add colon for pseudocode format
        lines.add(headerLine);

        //------- Comment generation -------
        // Generates a comment based on the method name to indicate its functionality
        // base on CRUD operations.
        String comment = "        // ... ";
        // We extract the method name for analysis (before the '(').
        int parenIndex = processedMethodSignature.indexOf("(");
        String methodName = (parenIndex != -1) ? processedMethodSignature.substring(0, parenIndex).trim()
: processedMethodSignature; // Extract method name

        // Generate comments based on CRUD operations and class type
        if (methodName.startsWith("create")) {
            if (className.endsWith("_table")) {
                String entityName = className.substring(0, className.indexOf("_table")).toLowerCase();
                comment += "Insert " + entityName + " record into the database";
            } else {
                comment += "Creation logic here";
            }
        } else if (methodName.startsWith("read")) {
            if (className.endsWith("_table")) {
                String entityName = className.substring(0, className.indexOf("_table")).toLowerCase();
                comment += "Retrieve " + entityName + " record from the database";
            } else {
                comment += "Retrieval logic here";
            }
        } else if (methodName.startsWith("update")) {
            if (className.endsWith("_table")) {
                String entityName = className.substring(0, className.indexOf("_table")).toLowerCase();
                comment += "Update " + entityName + " record in the database";
            } else {
                comment += "Update logic here";
            }
        } else if (methodName.startsWith("delete")) {
            if (className.endsWith("_table")) {
                String entityName = className.substring(0, className.indexOf("_table")).toLowerCase();
                comment += "Delete " + entityName + " record from the database";
            } else {
                comment += "Deletion logic here";
            }
        } else if (methodName.startsWith("drive")) {
            comment += "Logic to simulate driving";
        } else {
            comment += "Method logic here";
        }
```



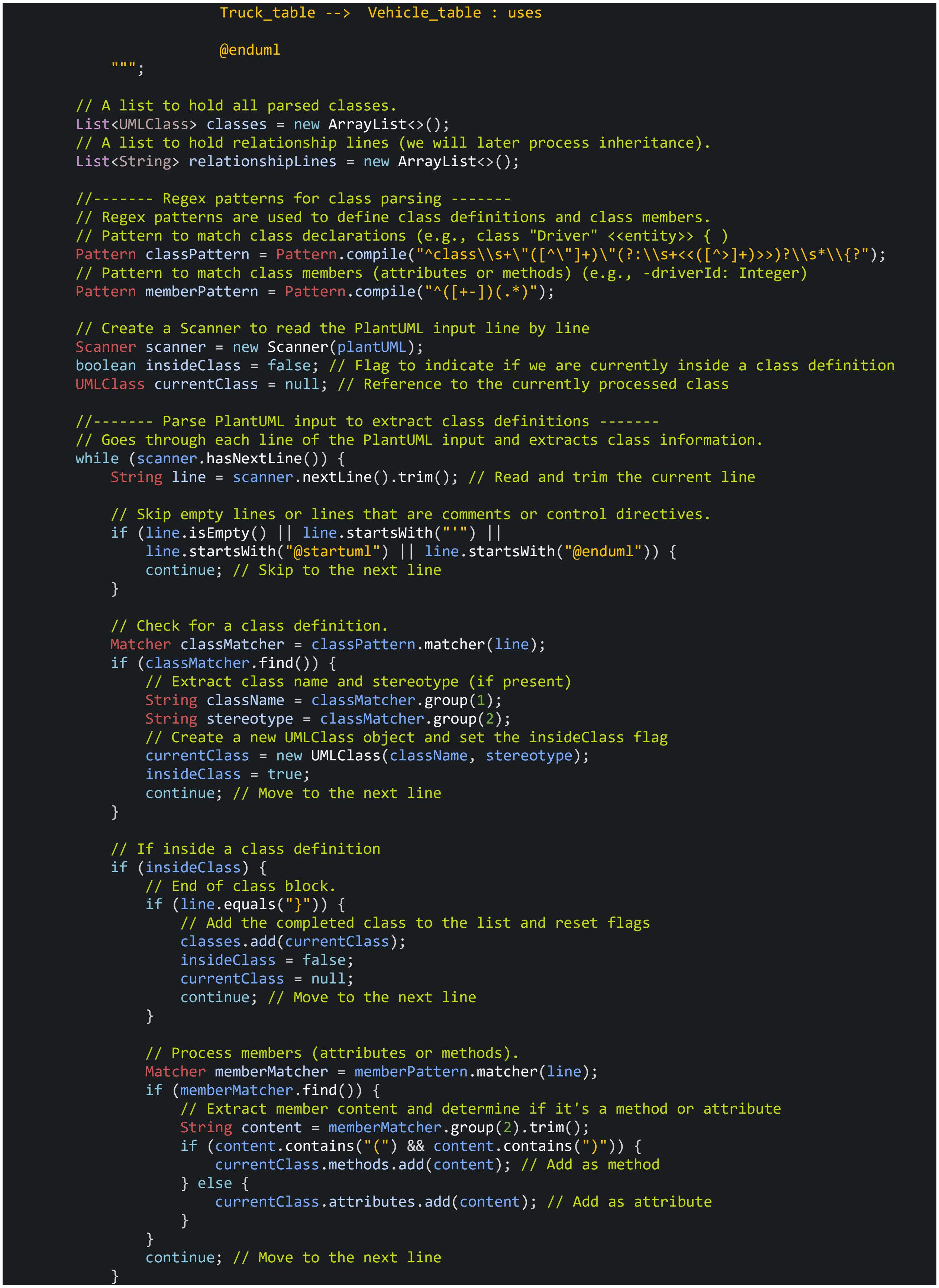
```java
        lines.add(comment);

        //------- Return statement -------
        // Builds the return statement line based on the method's return type.
        String returnLine = "        return ";
        if (returnType.equalsIgnoreCase("Void") || returnType.isEmpty()) {
            returnLine += "void"; // Return void for void methods
        } else {
            returnLine += returnType; // Return the specified type
        }
        lines.add(returnLine);

        return lines;
    }

    // -------------------------------------------------------------------------------------------------

    /**
     * Splits the parameter string into individual parameters
     * while keeping commas inside square brackets intact.
     * (e.g. Array[Integer, Integer])
     *
     * @param params The parameter list as a string.
     * @return A list of parameter strings.
     */
    private static List<String> splitParameters(String params) {
        List<String> result = new ArrayList<>();
        StringBuilder current = new StringBuilder();
        int bracketLevel = 0;  // Track nesting level of brackets

        // Iterate through the parameter string
        for (int i = 0; i < params.length(); i++) {
            char c = params.charAt(i);
            if (c == '[') {
                bracketLevel++;  // Increase level at opening bracket
            } else if (c == ']') {
                if (bracketLevel > 0) {
                    bracketLevel--;  // Decrease level at closing bracket
                }
            } else if (c == ',' && bracketLevel == 0) {
                // Split at comma only if not inside brackets
                result.add(current.toString());
                current.setLength(0); // Reset for next parameter
                continue;
            }
            current.append(c);
        }
        // Add the last parameter (if any)
        if (current.length() > 0) {
            result.add(current.toString());
        }
        return result;
    }

    // ---------------------------------------------------------------------------------------------------
    /*--------------------
    |    Main Method    |
    --------------------*/

    /**
     *  Main method: Parses the PlantUML diagram and outputs the pseudocode.
     *
     * @param args Command line arguments (not used in this program).
     */
    public static void main(String[] args) {
        // Text block of the The PlantUML code
        String plantUML = """
            @startuml
```



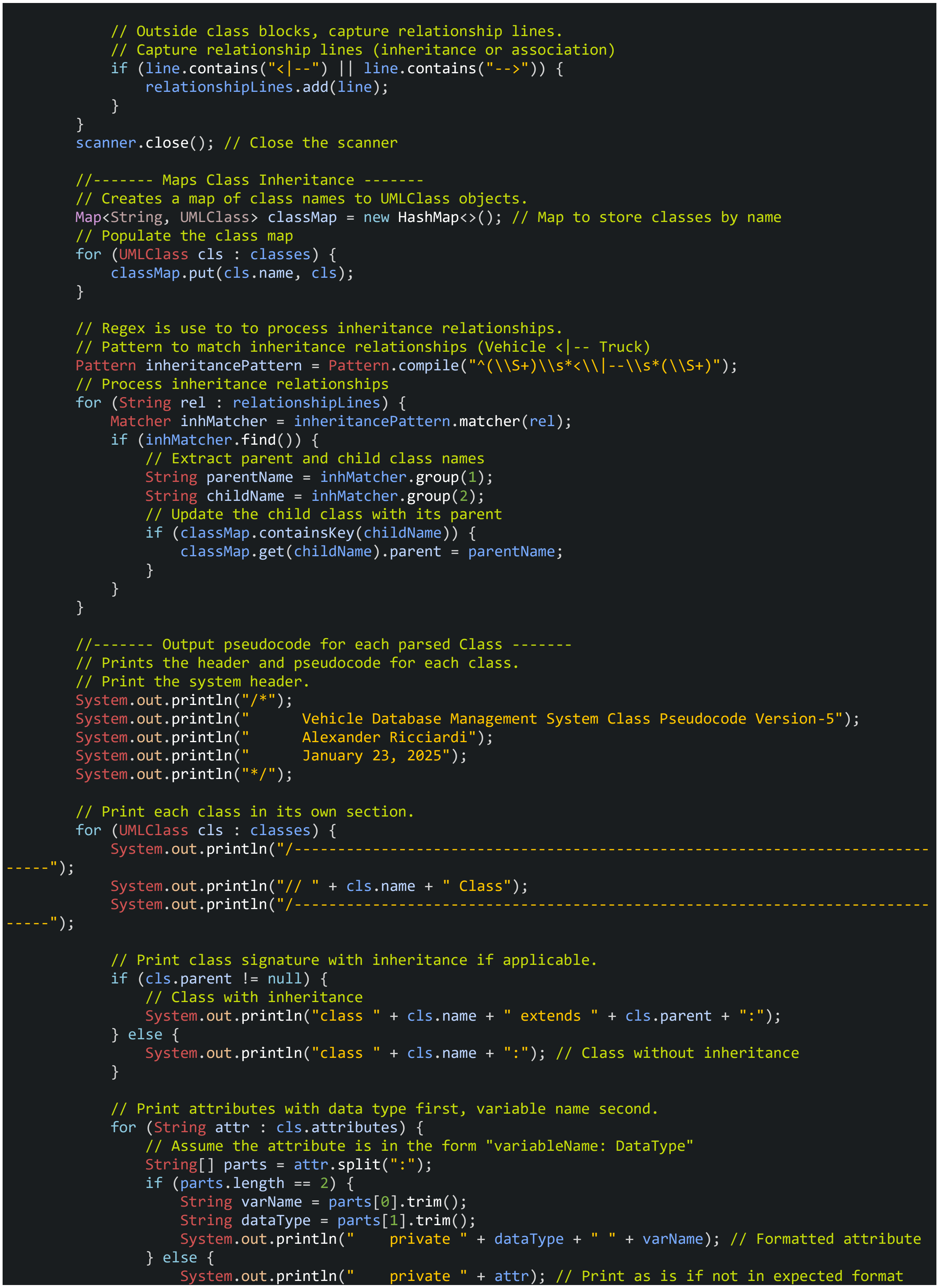
```
' Entity Classes
'
class "Driver" <<entity>> {
  -driverId: Integer
  -lastName: String
  -cars: List<Car>
}

class "Vehicle" <<entity>> {
  -vehicleId: Integer
  -vinNum: String
  -driverIds: List<Integer>
  +drive()
}

class "Truck" <<entity>> {
  -truckId: Integer
  -truckType: String
}

class "Car" <<entity>> {
  -carId: Integer
  -carType: String
}

'
' Control Class
'
class "DriverCarManager" <<control>> {
+createDriver(driverId: Integer, driverDtls): Void
+readDriver(driverId: Integer): driverDtls
+updateDriver(driverId: Integer, driverDtls): Void
+deleteDriver(driverId: Integer): Void

+createVehicle(vehiculeId: Integer, vehicleDtls): Void
+readVehicle(vehicleId: Integer): vehicleDtls
+updateVehicle(carId: Integer): Void
+deleteVehicle(carId: Integer): Void

+createCar(carId: Integer, carDtls): Void
+readCar(carId: Integer): carDtls
+updateCar(carId: Integer, carDtls): Void
+deleteCar(carId: Integer): Void

+createTruck(truckId: Integer, truckDtls): Void
+readTruck(truckId: Integer): truckDtls
+updateTruck(truckId: Integer, truckDtls): Void
+deleteTruck(truckId: Integer): Void

+createDriverVehicle(driverVehicleId: Array[Integer, Integer], driverVehicleDtls): Void
+readDriverVehicle(driverVehicleId: Integer): driverVehicleDtls
+updateDriverVehicle(driverVehicleId: Integer, driverVehicleDtls): Void
+deleteDriverVehicle(driverVehicleId: Integer): Void
}

'
' Table Classes
'
class "Driver_table" <<table>> {
  -driverId: Integer
  -driverName: String

  +createDriver(driverId: Integer, driverDtls): Void
  +readDriver(driverId: Integer): driverDtls
  +updateDriver(driverId: Integer, driverDtls): Void
  +deleteDriver(driverId: Integer): Void
}
```



```
                         class "Vehicle_table" <<table>> {
                           -vehicleId: Integer
                           -vinNum: String

                           +createVehicle(vehicleId: Integer, vehicleDtls): Void
                           +readVehicle(vehicleId: Integer): vehicleDtls
                           +updateVehicle(carId: Integer): Void
                           +deleteVehicle(carId: Integer): Void
                         }

                         class "Driver_Vehicle_table" <<table>> {
                           -driverVehicleId: [Integer, Integer]
                           -driverId: Integer
                           -vehicleId: Integer

                           +createDriverVehicle(driverVehicleId: Array[Integer, Integer],
driverVehicleDtls): Void
                           +readDriverVehicle(driverVehicleId: Integer): driverVehicleDtls
                           +updateDriverVehicle(driverVehicleId: Integer, driverVehicleDtls): Void
                           +deleteDriverVehicle(driverVehicleId: Integer): Void
                         }

                         class "Truck_table" <<table>> {
                           -truckId: Integer
                           -truckType: String

                           +createTruck(truckId: Integer, truckDtls): Void
                           +readTruck(truckId: Integer): truckDtls
                           +updateTruck(truckId: Integer, truckDtls): Void
                           +deleteTruck(truckId: Integer): Void
                         }

                         class "Car_table" <<table>> {
                           -carId: Integer
                           -carType: String

                           +createCar(carId: Integer, carDtls): Void
                           +readCar(carId: Integer): carDtls
                           +updateCar(carId: Integer, carDtls): Void
                           +deleteCar(carId: Integer): Void
                         }

                         '
                         ' Relationships
                         '
                         ' Inheritance (Vehicle is a superclass of Truck and Car)
                         Vehicle <|-- Truck
                         Vehicle <|-- Car

                         ' Driver can drive many Vehicles (1..N)
                         Driver "1" --> "1..N" Vehicle : drives

                         ' Driver and Vehicle both use the DriverCarManager
                         Driver "1..N" --> "1" DriverCarManager : uses
                         Vehicle "1..N" --> "1" DriverCarManager : uses

                         ' DriverCarManager uses the database tables
                         DriverCarManager --> Driver_table : uses
                         DriverCarManager --> Vehicle_table : uses
                         DriverCarManager --> Car_table : uses
                         DriverCarManager --> Truck_table : uses
                         DriverCarManager --> Driver_Vehicle_table : uses

                         ' Driver_Vehicle_table use the Driver_table and Vehicle_table
                         Driver_Vehicle_table --> Driver_table : uses
                         Driver_Vehicle_table --> Vehicle_table : uses

                         ' Car_table and Truck_table use the Vehicle_table
                         Car_table -->  Vehicle_table : uses
```



```
                        Truck_table -->  Vehicle_table : uses

                        @enduml
            """;

        // A list to hold all parsed classes.
        List<UMLClass> classes = new ArrayList<>();
        // A list to hold relationship lines (we will later process inheritance).
        List<String> relationshipLines = new ArrayList<>();

        //------- Regex patterns for class parsing -------
        // Regex patterns are used to define class definitions and class members.
        // Pattern to match class declarations (e.g., class "Driver" <<entity>> { )
        Pattern classPattern = Pattern.compile("^class\\s+\"([^\"]+)\"(?:\\s+<<([^>]+)>>)?\\s*\\{?");
        // Pattern to match class members (attributes or methods) (e.g., -driverId: Integer)
        Pattern memberPattern = Pattern.compile("^([+-])(.*)");

        // Create a Scanner to read the PlantUML input line by line
        Scanner scanner = new Scanner(plantUML);
        boolean insideClass = false; // Flag to indicate if we are currently inside a class definition
        UMLClass currentClass = null; // Reference to the currently processed class

        //------- Parse PlantUML input to extract class definitions -------
        // Goes through each line of the PlantUML input and extracts class information.
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine().trim(); // Read and trim the current line

            // Skip empty lines or lines that are comments or control directives.
            if (line.isEmpty() || line.startsWith("'") ||
                line.startsWith("@startuml") || line.startsWith("@enduml")) {
                continue; // Skip to the next line
            }

            // Check for a class definition.
            Matcher classMatcher = classPattern.matcher(line);
            if (classMatcher.find()) {
                // Extract class name and stereotype (if present)
                String className = classMatcher.group(1);
                String stereotype = classMatcher.group(2);
                // Create a new UMLClass object and set the insideClass flag
                currentClass = new UMLClass(className, stereotype);
                insideClass = true;
                continue; // Move to the next line
            }

            // If inside a class definition
            if (insideClass) {
                // End of class block.
                if (line.equals("}")) {
                    // Add the completed class to the list and reset flags
                    classes.add(currentClass);
                    insideClass = false;
                    currentClass = null;
                    continue; // Move to the next line
                }

                // Process members (attributes or methods).
                Matcher memberMatcher = memberPattern.matcher(line);
                if (memberMatcher.find()) {
                    // Extract member content and determine if it's a method or attribute
                    String content = memberMatcher.group(2).trim();
                    if (content.contains("(") && content.contains(")")) {
                        currentClass.methods.add(content); // Add as method
                    } else {
                        currentClass.attributes.add(content); // Add as attribute
                    }
                }
                continue; // Move to the next line
            }
```

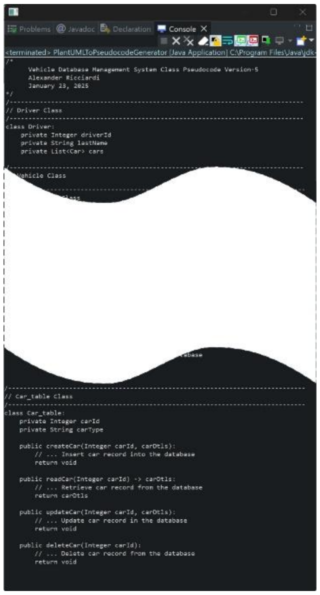


```
            // Outside class blocks, capture relationship lines.
            // Capture relationship lines (inheritance or association)
            if (line.contains("<|--") || line.contains("-->")) {
                relationshipLines.add(line);
            }
        }
        scanner.close(); // Close the scanner

        //------- Maps Class Inheritance -------
        // Creates a map of class names to UMLClass objects.
        Map<String, UMLClass> classMap = new HashMap<>(); // Map to store classes by name
        // Populate the class map
        for (UMLClass cls : classes) {
            classMap.put(cls.name, cls);
        }

        // Regex is use to to process inheritance relationships.
        // Pattern to match inheritance relationships (Vehicle <|-- Truck)
        Pattern inheritancePattern = Pattern.compile("^(\\S+)\\s*<\\|--\\s*(\\S+)");
        // Process inheritance relationships
        for (String rel : relationshipLines) {
            Matcher inhMatcher = inheritancePattern.matcher(rel);
            if (inhMatcher.find()) {
                // Extract parent and child class names
                String parentName = inhMatcher.group(1);
                String childName = inhMatcher.group(2);
                // Update the child class with its parent
                if (classMap.containsKey(childName)) {
                    classMap.get(childName).parent = parentName;
                }
            }
        }

        //------- Output pseudocode for each parsed Class -------
        // Prints the header and pseudocode for each class.
        // Print the system header.
        System.out.println("/*");
        System.out.println("      Vehicle Database Management System Class Pseudocode Version-5");
        System.out.println("      Alexander Ricciardi");
        System.out.println("      January 23, 2025");
        System.out.println("*/");

        // Print each class in its own section.
        for (UMLClass cls : classes) {
            System.out.println("/--------------------------------------------------------------------------
-----");
            System.out.println("// " + cls.name + " Class");
            System.out.println("/--------------------------------------------------------------------------
-----");

            // Print class signature with inheritance if applicable.
            if (cls.parent != null) {
                // Class with inheritance
                System.out.println("class " + cls.name + " extends " + cls.parent + ":");
            } else {
                System.out.println("class " + cls.name + ":"); // Class without inheritance
            }

            // Print attributes with data type first, variable name second.
            for (String attr : cls.attributes) {
                // Assume the attribute is in the form "variableName: DataType"
                String[] parts = attr.split(":");
                if (parts.length == 2) {
                    String varName = parts[0].trim();
                    String dataType = parts[1].trim();
                    System.out.println("    private " + dataType + " " + varName); // Formatted attribute
                } else {
                    System.out.println("    private " + attr); // Print as is if not in expected format
                }
            }

            // Add a blank line if there are both attributes and methods.
            if (!cls.attributes.isEmpty() && !cls.methods.isEmpty()) {
                System.out.println(); // Blank line for separation
            }

            // Print methods using formatting helper.
            for (String method : cls.methods) {
                // Generate pseudocode
                List<String> methodLines = generateMethodPseudocode(method, cls.name);
                for (String lineOut : methodLines) {
                    System.out.println(lineOut); // Print each line of pseudocode
                }
                System.out.println(); // Blank line after each method.
            }
            System.out.println(); // Blank line after each class.
        }
    }
}
```

Note: the code snippet contains the code lines of the PlantUML Class Diagram To Pseudocode Generator, which is a small program that translates PlantUML class diagram code To pseudocode. Note that the PlantUML code from the vehicle database management system class diagram was integrated into the code as a multi-line literal string.

Figure 11

PlantUML Class Diagram To Pseudocode Generator Version-5



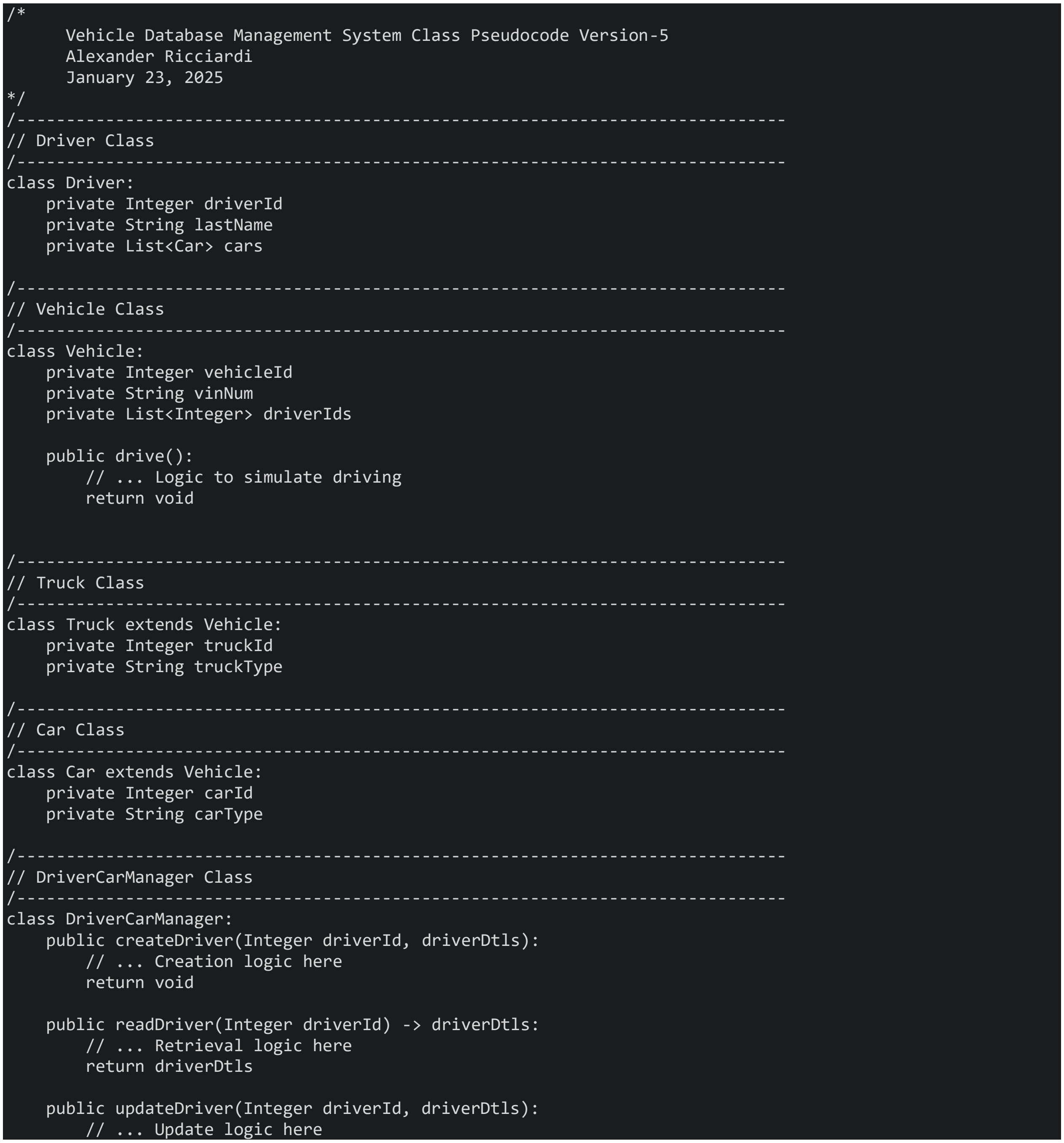
```
<terminated> PlantUMLToPseudocodeGenerator [Java Application] C:\Program Files\Java\jdk-
/*
     Vehicle Database Management System Class Pseudocode Version-5
     Alexander Ricciardi
     January 23, 2025
*/
/--------------------------------------------------------------------------------
// Driver Class
/--------------------------------------------------------------------------------
class Driver:
    private Integer driverId
    private String lastName
    private List<Car> cars

/--------------------------------------------------------------------------------
  Vehicle Class

/--------------------------------------------------------------------------------
// Car_table Class
/--------------------------------------------------------------------------------
class Car_table:
    private Integer carId
    private String carType

    public createCar(Integer carId, carDtls):
        // ... Insert car record into the database
        return void

    public readCar(Integer carId) -> carDtls:
        // ... Retrieve car record from the database
        return carDtls

    public updateCar(Integer carId, carDtls):
        // ... Update car record in the database
        return void

    public deleteCar(Integer carId):
        // ... Delete car record from the database
        return void
```

Note: The figure illustrates the translated PlandUML code to pseudocode of the vehicle database management system class diagram. Note that the picture does not depict the full code.

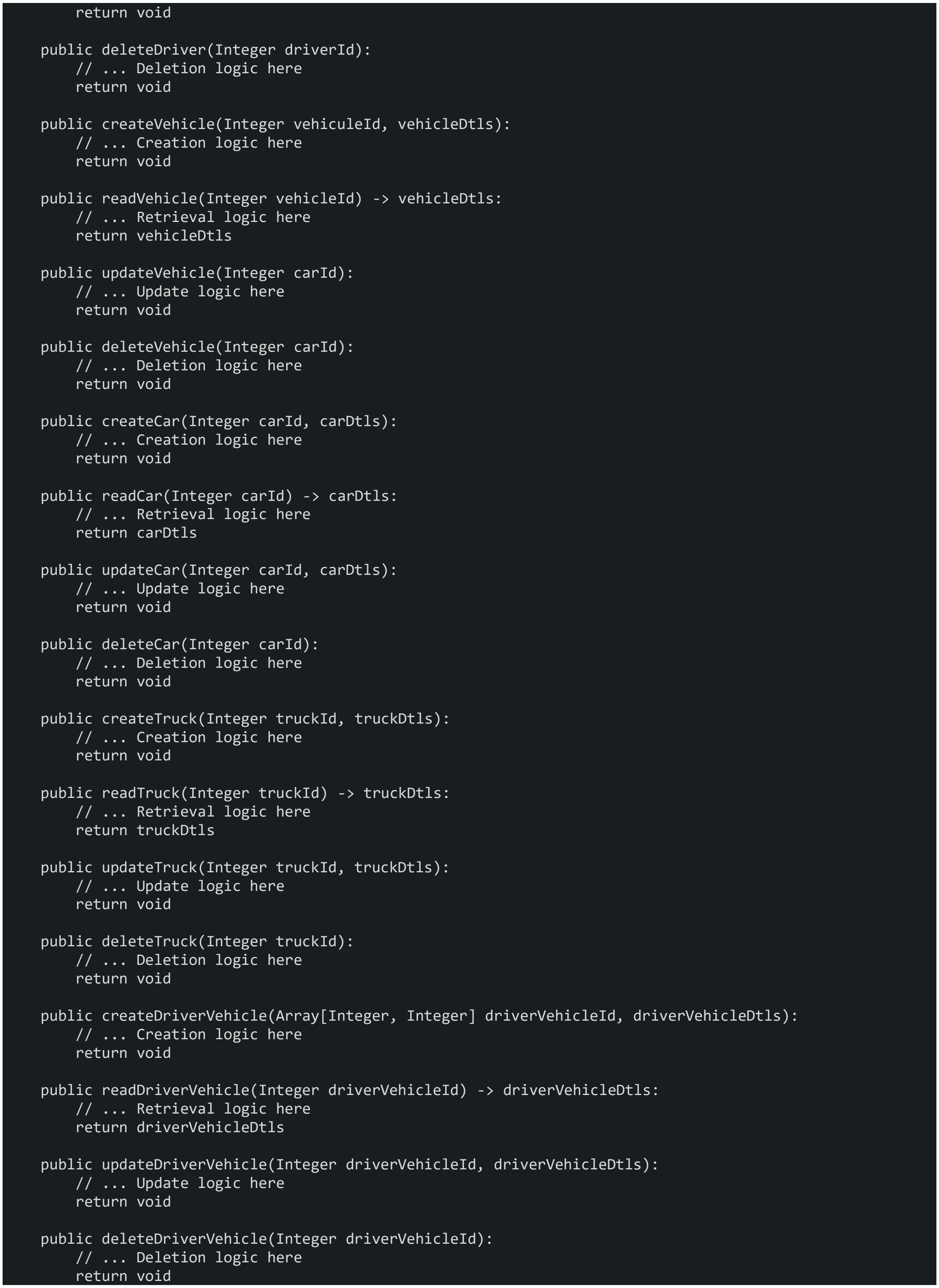
Figure 11 illustrates the output from the Java code in the console. The output is the translated PlantUML UML class diagram to pseudocode. The code snippet below is the actual output printed in the console.

Code Snippet 4

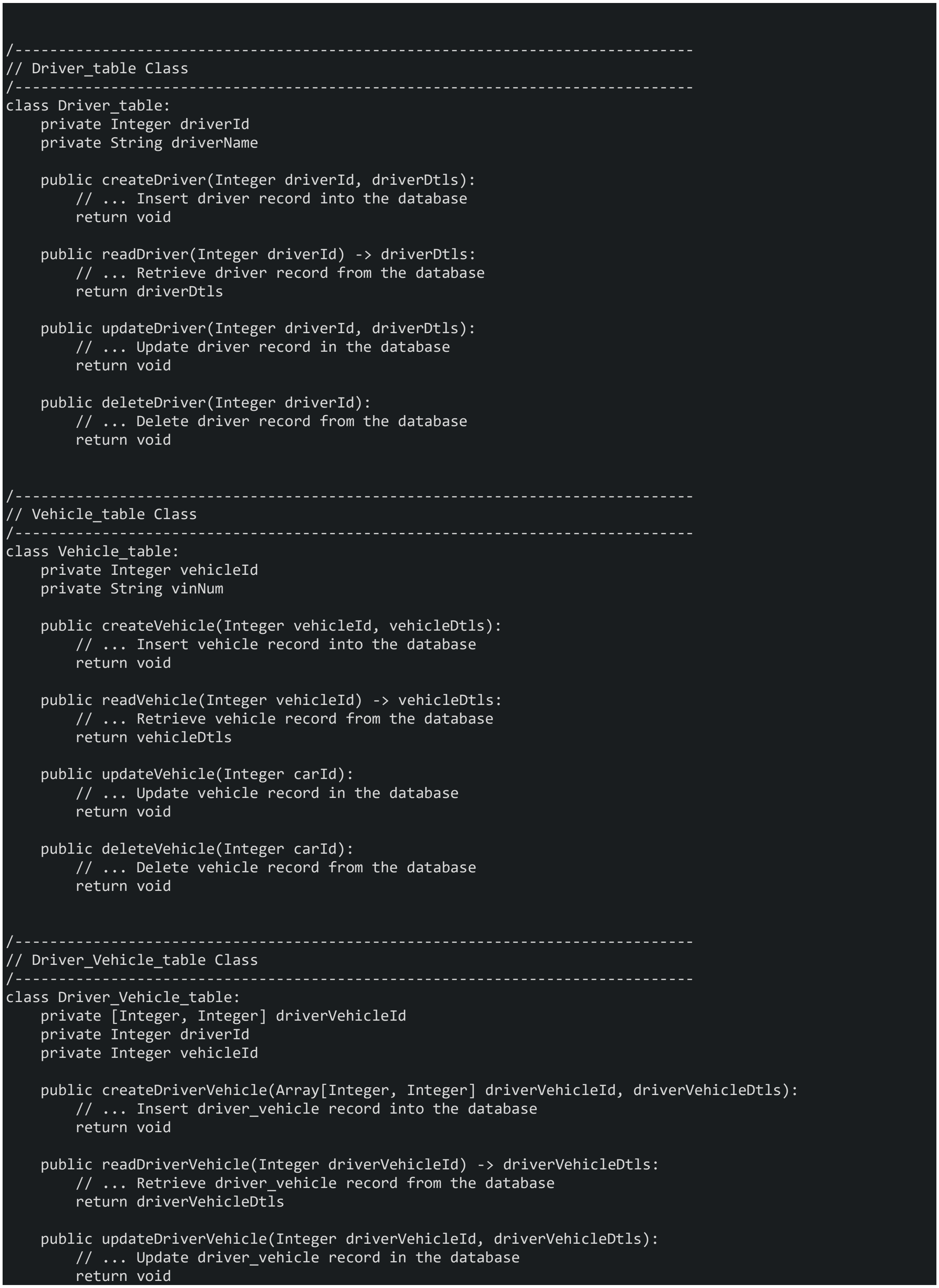
Pseudocode Output From the PlantUML Class Diagram To Pseudocode Generator Program



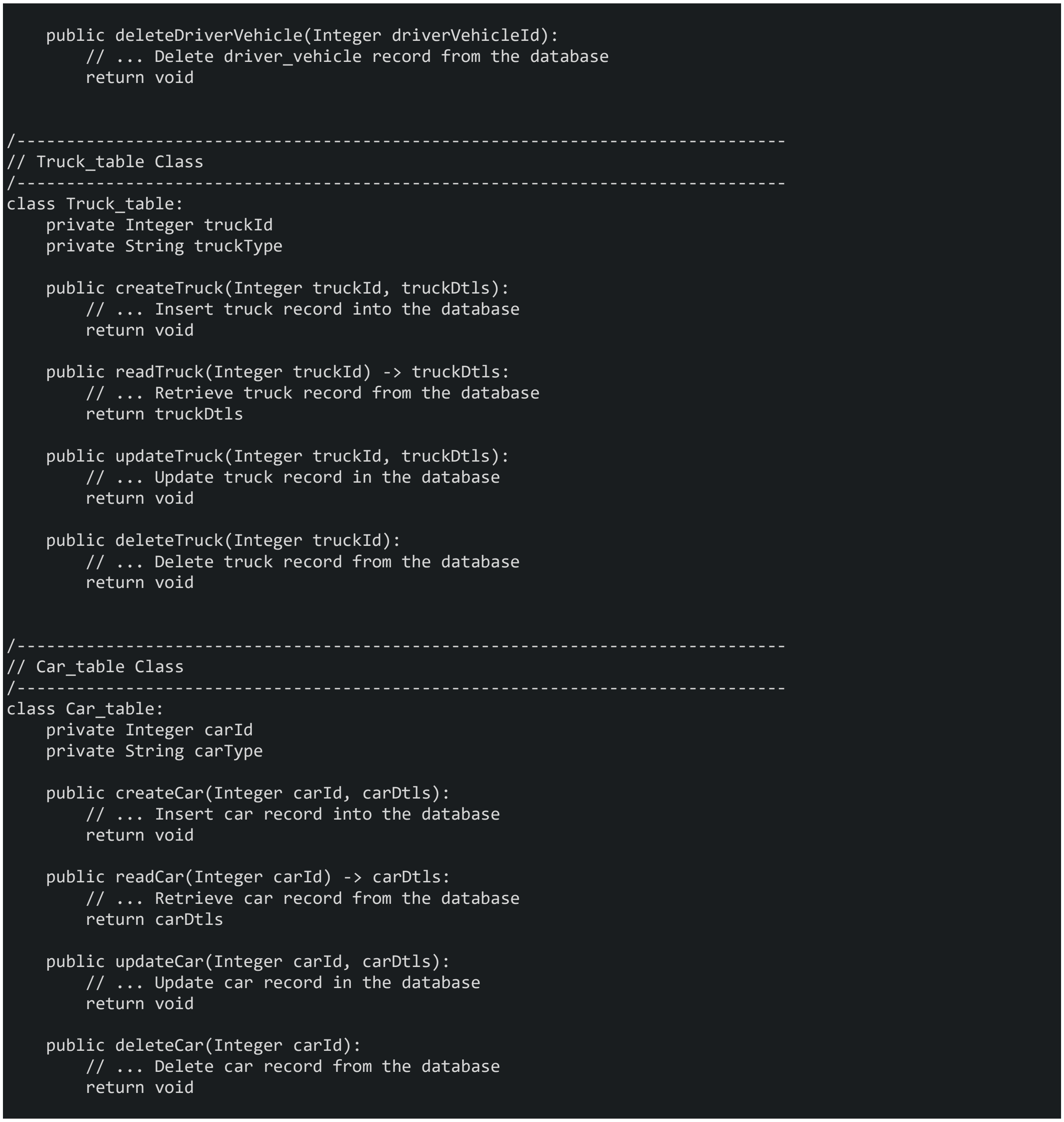
```
/*
      Vehicle Database Management System Class Pseudocode Version-5
      Alexander Ricciardi
      January 23, 2025
*/
/------------------------------------------------------------------------------
// Driver Class
/------------------------------------------------------------------------------
class Driver:
    private Integer driverId
    private String lastName
    private List<Car> cars

/------------------------------------------------------------------------------
// Vehicle Class
/------------------------------------------------------------------------------
class Vehicle:
    private Integer vehicleId
    private String vinNum
    private List<Integer> driverIds

    public drive():
        // ... Logic to simulate driving
        return void


/------------------------------------------------------------------------------
// Truck Class
/------------------------------------------------------------------------------
class Truck extends Vehicle:
    private Integer truckId
    private String truckType

/------------------------------------------------------------------------------
// Car Class
/------------------------------------------------------------------------------
class Car extends Vehicle:
    private Integer carId
    private String carType

/------------------------------------------------------------------------------
// DriverCarManager Class
/------------------------------------------------------------------------------
class DriverCarManager:
    public createDriver(Integer driverId, driverDtls):
        // ... Creation logic here
        return void

    public readDriver(Integer driverId) -> driverDtls:
        // ... Retrieval logic here
        return driverDtls

    public updateDriver(Integer driverId, driverDtls):
        // ... Update logic here
```



```
        return void

    public deleteDriver(Integer driverId):
        // ... Deletion logic here
        return void

    public createVehicle(Integer vehiculeId, vehicleDtls):
        // ... Creation logic here
        return void

    public readVehicle(Integer vehicleId) -> vehicleDtls:
        // ... Retrieval logic here
        return vehicleDtls

    public updateVehicle(Integer carId):
        // ... Update logic here
        return void

    public deleteVehicle(Integer carId):
        // ... Deletion logic here
        return void

    public createCar(Integer carId, carDtls):
        // ... Creation logic here
        return void

    public readCar(Integer carId) -> carDtls:
        // ... Retrieval logic here
        return carDtls

    public updateCar(Integer carId, carDtls):
        // ... Update logic here
        return void

    public deleteCar(Integer carId):
        // ... Deletion logic here
        return void

    public createTruck(Integer truckId, truckDtls):
        // ... Creation logic here
        return void

    public readTruck(Integer truckId) -> truckDtls:
        // ... Retrieval logic here
        return truckDtls

    public updateTruck(Integer truckId, truckDtls):
        // ... Update logic here
        return void

    public deleteTruck(Integer truckId):
        // ... Deletion logic here
        return void

    public createDriverVehicle(Array[Integer, Integer] driverVehicleId, driverVehicleDtls):
        // ... Creation logic here
        return void

    public readDriverVehicle(Integer driverVehicleId) -> driverVehicleDtls:
        // ... Retrieval logic here
        return driverVehicleDtls

    public updateDriverVehicle(Integer driverVehicleId, driverVehicleDtls):
        // ... Update logic here
        return void

    public deleteDriverVehicle(Integer driverVehicleId):
        // ... Deletion logic here
        return void
```



```
/------------------------------------------------------------------------------
// Driver_table Class
/------------------------------------------------------------------------------
class Driver_table:
    private Integer driverId
    private String driverName

    public createDriver(Integer driverId, driverDtls):
        // ... Insert driver record into the database
        return void

    public readDriver(Integer driverId) -> driverDtls:
        // ... Retrieve driver record from the database
        return driverDtls

    public updateDriver(Integer driverId, driverDtls):
        // ... Update driver record in the database
        return void

    public deleteDriver(Integer driverId):
        // ... Delete driver record from the database
        return void


/------------------------------------------------------------------------------
// Vehicle_table Class
/------------------------------------------------------------------------------
class Vehicle_table:
    private Integer vehicleId
    private String vinNum

    public createVehicle(Integer vehicleId, vehicleDtls):
        // ... Insert vehicle record into the database
        return void

    public readVehicle(Integer vehicleId) -> vehicleDtls:
        // ... Retrieve vehicle record from the database
        return vehicleDtls

    public updateVehicle(Integer carId):
        // ... Update vehicle record in the database
        return void

    public deleteVehicle(Integer carId):
        // ... Delete vehicle record from the database
        return void


/------------------------------------------------------------------------------
// Driver_Vehicle_table Class
/------------------------------------------------------------------------------
class Driver_Vehicle_table:
    private [Integer, Integer] driverVehicleId
    private Integer driverId
    private Integer vehicleId

    public createDriverVehicle(Array[Integer, Integer] driverVehicleId, driverVehicleDtls):
        // ... Insert driver_vehicle record into the database
        return void

    public readDriverVehicle(Integer driverVehicleId) -> driverVehicleDtls:
        // ... Retrieve driver_vehicle record from the database
        return driverVehicleDtls

    public updateDriverVehicle(Integer driverVehicleId, driverVehicleDtls):
        // ... Update driver_vehicle record in the database
        return void
```



```
    public deleteDriverVehicle(Integer driverVehicleId):
        // ... Delete driver_vehicle record from the database
        return void


/-------------------------------------------------------------------------------
// Truck_table Class
/-------------------------------------------------------------------------------
class Truck_table:
    private Integer truckId
    private String truckType

    public createTruck(Integer truckId, truckDtls):
        // ... Insert truck record into the database
        return void

    public readTruck(Integer truckId) -> truckDtls:
        // ... Retrieve truck record from the database
        return truckDtls

    public updateTruck(Integer truckId, truckDtls):
        // ... Update truck record in the database
        return void

    public deleteTruck(Integer truckId):
        // ... Delete truck record from the database
        return void


/-------------------------------------------------------------------------------
// Car_table Class
/-------------------------------------------------------------------------------
class Car_table:
    private Integer carId
    private String carType

    public createCar(Integer carId, carDtls):
        // ... Insert car record into the database
        return void

    public readCar(Integer carId) -> carDtls:
        // ... Retrieve car record from the database
        return carDtls

    public updateCar(Integer carId, carDtls):
        // ... Update car record in the database
        return void

    public deleteCar(Integer carId):
        // ... Delete car record from the database
        return void
```

Note: The code snippet contains the pseudocode lines outputted from the PlantUML class diagram to the pseudocode generator program. The pseudocode represents the vehicle database management system classes depicted by the PlantUML class diagram.

As shown by the figures and the code snippets in this section, especially Code Snippet 4, the Java program successfully translated the PlantUML code of the vehicle database management system class diagram. Additionally, the pseudocode accurately represents the classes depicted in Figure 8, 'Vehicle Database Management System Class Diagram,' from Part 1 of this paper.

Part-3 Reflection Lessons Learned (CSC470)

The winter semester B Software Engineering (CS470) course at CSU Global has introduced me to software engineering concepts and given me skills that I can apply software to model and develop software. The course emphasized the evaluation of system requirements and the analysis of software through the use of Unified Modeling Language (UML) diagrams. The course has also provided me with a good understanding of the entire software development lifecycle, from requirements gathering to deployment and maintenance. In this reflection, I will explore the key lessons that I learned and how they can apply and be applied toward more effective software engineering practices and modeling.

Before taking this course I had a layperson understanding of software development, I was aware that diagrams were used to model systems. However, I did not how they were used within the software development system. Additionaly, the only diagram types that I had some knowledge of how to use were flow charts and UML class diagrams. Furthermore, I was underestimating the crucial role that modeling plays in software development. The course taught me that modeling is not just about creating diagrams to understand a process; it's about creating shareable material that models, analyzes, and designs entire systems using various types of diagrams and methodologies such as Agile. Moreover, UML, provides diverse diagram types that have distinct purposes that can be used to capture various aspects of a system such as user interactions (use case diagrams), object structure and relationships (class diagrams), dynamic behavior and object interactions (sequence and state machine diagrams), and system workflows (activity diagrams). All these diagrams work together to analyze, model, and design systems providing a blueprint of the entire software lifecycle of those systems.

In my future software engineering endeavors, I will prioritize modeling, using UML diagrams that include:

- Use case diagrams which are the starting point for understanding user needs.
- Class diagrams that provide the foundation for designing object structures, which I will apply before coding.
- Sequence diagrams visualize object interactions, which I will use to identify design flaws early.
- Activity diagrams model describe the workflow behavior and dynamic aspect of systems, which I will use to model workflows and business processes.
- State machine diagrams visualize how objects change state over time, which I will use to model the dynamic behavior of individual objects within the system.

UML incorporates additional diagram types that can be used through the software lifecycle to analyze and design systems. Combining UML diagrams with a strong understanding of Object-Oriented principles, database modeling, and methodologies such as Agile, all concepts taught in the Software Engineering (CS470) course, will enable me to implement more effective software engineering practices and modeling in my professional life.

In conclusion, the CSC470 Software Engineering course gave me a new and more detailed understanding of software development. It has allowed me to transition from a novice programmer with limited UML diagram knowledge and understanding to a better-equipped junior software engineer with a good understanding of the entire software development lifecycle. In my future career as a software engineer, the concept of systems analysis, modeling, and design using UML, combined with an understanding of object-oriented principles, database systems, and agile development practices, will allow me to contribute effectively to project teams and deliver high-quality software.

References

Adservio (2021, March 15). *What are the pros and cons of NoSQL.* Adservio. https://www.adservio.fr/post/what-are-the-pros-and-cons-of-nosql

Arnold, J. (2023, August 30). Data consistency vs data integrity: Similarities and differences. IBM. https://www.ibm.com/think/topics/data-consistency-vs-data-integrity#:~:text=Data%20consistency%20refers%20to%20the,database%20systems%2%20applications%20and%20platforms.

Claris (2024). Many-to-many relationships. *Working with related tables*. Claris. https://help.claris.com/en/pro-help/content/many-to-many-relationships.html

Colorado State University Global (n.d.). *Module 7.1 Data Storage Mechanisms.* [Interactive lecture]. CSC470 Software Engineering, CSU Global, Departement of Computer Science. Canvas. Retrieved January 29, 2025, from: https://csuglobal.instructure.com/courses/104036/pages/7-dot-1-data-storage-mechanisms?module_item_id=5372300

CSU Global (n.d.). Module 8: Portfolio project [Assignment]. CSC470 Software Engineering, CSU Global, Department of Computer Science. Canvas. Retrieved from: https://csuglobal.instructure.com/courses/104036/assignments/1951062?module_item_id=5372363

Demarest, G. (2025, January 15). *NoSQL cloud databases: Benefits and features explained. Aerospike*. https://aerospike.com/blog/nosql-cloud-databases-benefits/

Foote, K. D. (2022, November 17). *NoSQL Databases: Advantages and Disadvantages*. DATAVERSITY. https://www.dataversity.net/nosql-databases-advantages-and-disadvantages/

Galton, A., & Mizoguchi, R. (2009). *The water falls but the waterfall does not fall: New perspectives on objects, processes and events*. Applied Ontology, 4(2), 71–107. https://doi.org/10.3233/AO-2009-0067

Google (n.d.). *What is a relational database?* Google Cloud. https://cloud.google.com/learn/what-is-a-relational-database

Greeff, G. & Ghoshal, R. (2004, August) 5 - Business process and system modeling tools and packages. Practical e-manufacturing and supply chain management. Newes. p. 112-145. ISBN 9780750662727

IBM (n.d.a). *What is data modeling?* IBM Think. https://www.ibm.com/think/topics/data-modeling

IBM (2022b, December 12). *What is a NoSQL database?* IBM Think. https://www.ibm.com/think/topics/nosql-databases

IBM (2021c, March 3). Database relationships. IBM Documentation. https://www.ibm.com/docs/en/mam/7.6.0?topic=structure-database-relationships

Imperva (n.d.). *NoSQL injection*. Imperva. https://www.imperva.com/learn/application-security/nosql-injection/

InterSytems (n.d.a). *NoSQL databases explained: Advantages, types, and use cases.* InterSytems. https://www.intersystems.com/resources/nosql-databases-explained-advantages-types-and-use-cases/

InterSystems (n.d.b). *What is a relational database and why do you need one?* InterSystems. https://www.intersystems.com/resources/what-is-a-relational-database/

Macrometa, (n.d.) Chapter 2 - Advantages and disadvantages of NoSQL. *Distributed data*. Macrometa. https://www.macrometa.com/distributed-data/advantages-and-disadvantages-of-nosql

Microsoft (n.d.). *What is a relational database?* Microsoft Azure. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-relational-database

MongoDB (n.d.). Advantages of NoSQL databases. MogoDB. https://www.mongodb.com/resources/basics/databases/nosql-explained/advantages

Nalimov, C. (2024, April 16). *Exploring one-to-many relationship examples in databases*. Gleek. https://www.gleek.io/blog/one-to-many

Navlaniwesr (2024, July 9). Database federation - System design. GeeksForGeeks. https://www.geeksforgeeks.org/database-federation-system-design/

PlantUML (n.d.). *PlantUML at a glance.* PlantUML. https://plantuml.com/

Quach, S. (n.d.). *NoSQL databases: NoSQL databases: What it is, how it works and how is it different from SQL.* Knowi. https://www.knowi.com/blog/nosql-databases-what-it-is-how-it-works-and-how-is-it-different-from-sql/

Rajputtzdb (2024, February 28) *What is Object-Relational Mapping (ORM) in DBMS?* GeeksForGeeks. https://www.geeksforgeeks.org/what-is-object-relational-mapping-orm-in-dbms/

Rajpurohit, A. (2023, March 28). Understanding Federation in Databases: Definition, types and use cases. Akash Rajpurohit. https://akashrajpurohit.com/blog/understanding-federation-in-databases-definition-types-and-use-cases/?utm_source

Regex101 (n.d.). Rgex101 [Web App.]. Regex101. https://regex101.com/

Ricciardi, A. (2025a, February 1). *Object-Oriented principles in software engineering: An overview of OODBMS, RDBMS, and ORM techniques*. Omegapy - Code Chronicles. https://www.alexomegapy.com/post/object-oriented-principles-in-software-engineering-an-overview-of-oodbms-rdbms-and-orm-techniques

Ricciardi, A. (2025b, January 21). *UML class Diagrams: modeling systems from problem space to solution space*. Omegapy- Code Chronicles. https://www.alexomegapy.com/post/uml-class-diagrams-modeling-systems-from-problem-space-to-solution-space

ScyllaDB (n.d.). *Database scalability*. ScyllaDB. https://www.scylladb.com/glossary/database-scalability/

Unhelkar, B. (2018). Chapter 13 — Database modeling with class and sequence Diagrams. *Software engineering with UML*. CRC Press. ISBN 9781138297432

Virgo, J. (2025, February 13). *Relationships in SQL – complete guide with examples*. Dittofi. https://www.dittofi.com/learn/relationships-in-sql-complete-guide-with-examples