

Instructions

This project was made using Java v14, Maven, and PostgreSQL. The correct versions can be found in the pom.xml file in the /uwtsd-taxolotl directory.

The database details are coded into the DatabaseControlller.java class, but are as follows for clarity:

```
DATABASE_USERNAME = "postgres";
```

```
DATABASE_PASSWORD = "password";
```

```
DATABASE_URL = "jdbc:postgresql://localhost:5432/";
```

To set up the database, there are two options. First, run taxolotl_schema.sql in your chosen IDE (IntelliJ includes a database manager, for example) and then the .sql files in the /db_dump folder. Alternatively, run taxolotl_schema.sql and then uncomment DatabaseSeeder.seedDatabase() in Main.java and run the program in order to seed your own data.

Database Design Reflection

Zack Costello

Student Number: 2012480

2012480@student.uwtsd.ac.uk

Module: ACCA7002

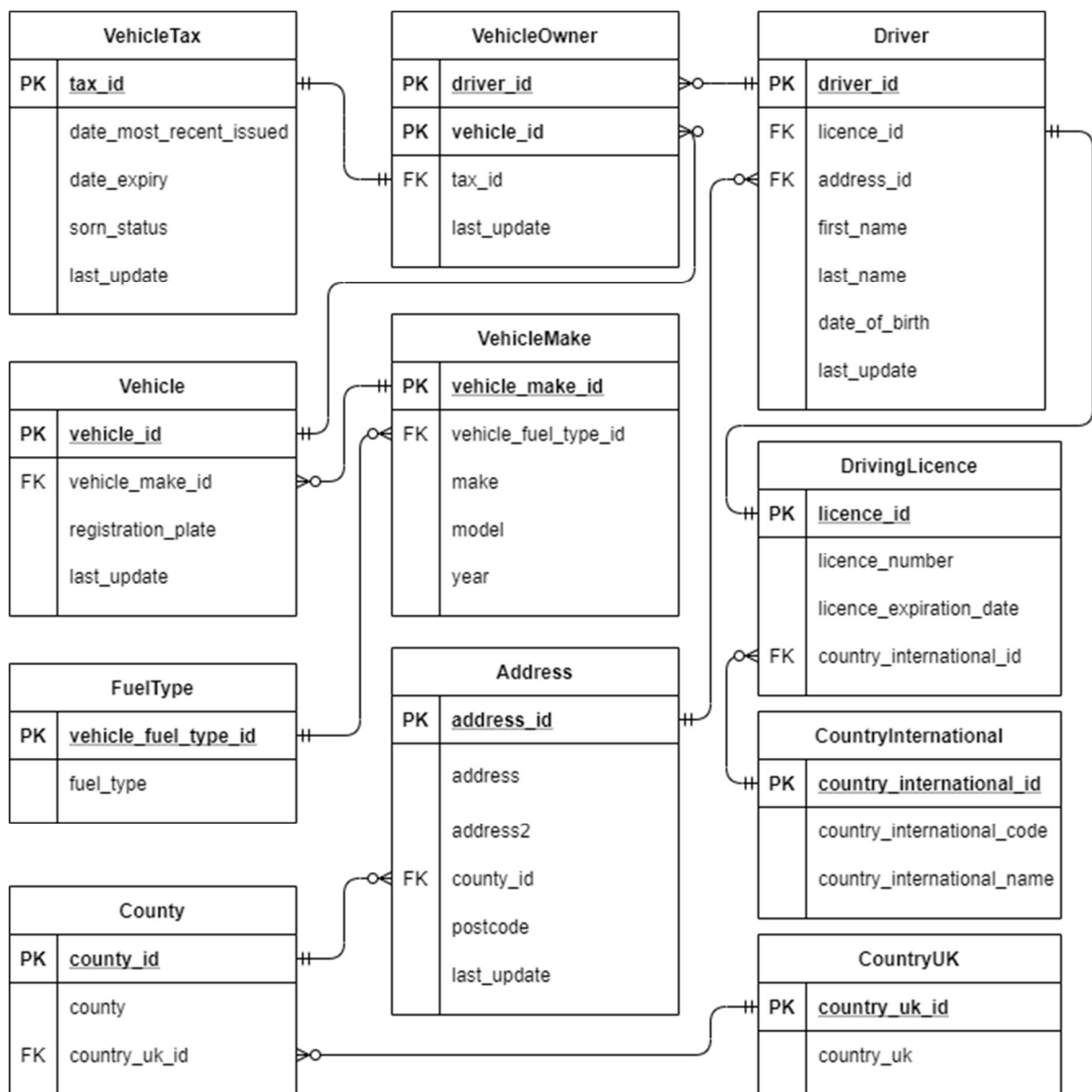


Figure 1: Entity-Relationship (ER) diagram for the car tax database

The ER diagram for the database is shown above in Fig. 1. The composite Primary Key (PK) was chosen to be in the VehicleOwner table due to the *driver_id* and *vehicle_id* being a unique combination by design.

The .sql schema that creates the tables for the database can be found in the /db-sql folder, along with a SQL dump from the database with seeded values. Within the schema, several check constraints were used:

- `check (driver.date_of_birth > '1900-01-01' and driver.date_of_birth <= (now() - interval '16 years'))`
Which ensures that the driver is born after 1900 and that they are at least 16 years old (as they would otherwise be unable to have a driving licence).
- `check (vehicle_tax.date_expiry = vehicle_tax.date_most_recent_issued + interval '12 months')`
Which ensures that the road tax expires exactly 12 months after the most recent licence has been issued.
- `check (vehicle.registration_plate ~ '[A-Z]{2}[0-9]{2}[A-Z]{3}$')`
Which performs a regular expression (RegEx) to ensure that only vehicle licence plates with the correct format can be inputted into the database.

As can be seen in the .sql files, more check constraints were used, but were of a similar nature to the examples shown above. The .gov website provided several RegExes that were used for ensuring the correct format for various licence numbers.

The unique constraint was also used in several places to ensure that no duplicates could be entered into the database, as well as the “on update cascade” modifier so that changes would be made in all connected tables.

Programming Design Reflection

Zack Costello

Student Number: 2012480

2012480@student.uwtsd.ac.uk

Module: ACCA7000

The Taxolotl database was, in hindsight, perhaps quite ambitious given the time constraints. Seeding the database took a considerable amount of time due to the complexity of the database and the difficulty in finding or generating adequate data to be used when generating fields. The database seeding was done using the Java Faker library, and the process can be seen in the DatabaseSeeder.java class.

A menu system was implemented, along with classes for formatting the query outputs into tables (with the sizes autogenerated). The interface with the database was completed, meaning the implementation of all of the features is possible in the future using the classes already developed.

Prepared statements were used for user queries, which help prevent SQL injection attacks and also improve query performance.

Try searching by driving licence using the following test inputs:

Z9SCX795066KC4FO

Z9L992346559MY13

Four Software Engineering Concepts and Their Importance

Zack Costello

Student Number: 2012480

2012480@student.uwtsd.ac.uk

Module: ACCA7000

Word Count (excluding Abstract, Headers, Tables, Figures, and References): 1645

Abstract

This report discusses the importance of four software design principles: high cohesion, loose coupling, data hiding, and information hiding. The principles are all first defined and then covered in more detail, being reinforced by relevant literature. Descriptions of different cohesion and coupling types are presented, along with examples that clarify the four concepts. The interdependency of these principles is shown and this is further explored by showing how they are normally considered together when assessing software design quality with reference to appropriate research.

I. INTRODUCTION

When learning the fundamentals of programming, or *Learning to Code*, it is common (and arguably necessary) to initially restrict the scope of learning to that of the individual, covering basic concepts such as loops and functions. However, it is unlikely that the individual will be utilising their newly-acquired skills in a vacuum and will instead be contributing to a project as part of a team, with the resultant increases in scope necessitating higher levels of code organisation to be considered. As such, common design methodologies which help ensure higher-quality software have been devised and adopted over the decades.

In addition, object-oriented programming (OOP), where code is organised into software classes (that act as the abstraction of some data) and objects (instances of a class which are structured data with associated properties and behaviour), is a programming paradigm that is a common feature of many widely-used programming languages [1]. OOP allows for many useful programming techniques (technical and organisational) that make it suitable for a wide variety of applications and is thus extensively used.

This report is concerned with the importance of four design principles fundamental to good software development: high cohesion, loose coupling, data hiding, and information hiding. These will primarily be described in the context of OOP for this report, but it should be noted that these concepts are just as applicable to (and were in fact originally devised for) other programming paradigms, particularly procedural programming.

II. HIGH COHESION

Cohesion is a characteristic dealing with how the different elements of a class are related to each other. A class is considered to have high cohesion if it consists of closely-related methods that deal with a central responsibility. Conversely, a class with low cohesion could be an illogically-grouped collection of general methods that bear little relation to each other [2].

There are different types of cohesion that have been identified and are typically placed on an ordinal scale. A short summary for several important types is given in Table I. It should be noted that the terminology and order of strength differs from source to source [1], [3]–[6]. However, all are consistent in noting that *functional* cohesion is the most desirable type and *temporal* cohesion is one of the least.

High cohesion is closely linked to the *Single Responsibility Principle* (SRP), which states that a class should have one, and only one, reason to change [7]. A relevant example is given in Fig. 1. The strongest type of cohesion,

TABLE I: Different Types of Cohesion, Ordered from Highest to Lowest in Terms of Cohesion Strength (adapted primarily from [6] but also [1] [5]).

Cohesion Type	Definition
Functional	Only <i>one functionality</i> is performed by a class
Sequential	A class is structured so methods <i>follow a sequence</i> (from input to output)
Communicational	A class only operates on the <i>same data</i>
Procedural	A class is constructed from a set of methods that are called <i>one after another</i>
Temporal	A class contains methods that are <i>related in time</i> (they are executed during roughly the same time or phase)
Logical	A class is structured so methods are <i>logically associated</i> , irrespective of their nature
Coincidental	A class structure has <i>little or no constructive relationships</i> between its elements

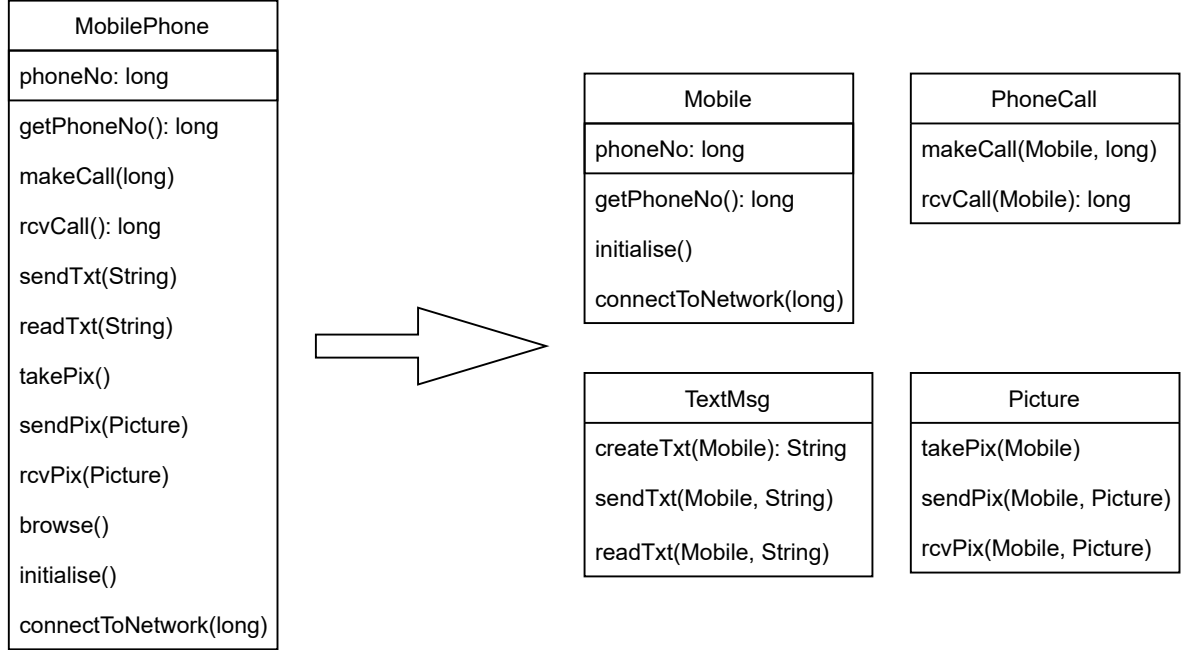


Fig. 1: An example `MobilePhone` class that violates the SRP and a solution that splits `MobilePhone` into smaller classes, each with a single responsibility (adapted from [8]).

functional, is clearly related to the SRP, ensuring that elements that belong together are kept together; the elements that belong together are those which facilitate only one functionality.

High cohesion encourages the use of many small classes, resulting in several advantages from a design perspective. Whilst it may initially seem like an increase in smaller classes would result in an increase in complexity, the inner mechanisms are often functionally the same as a larger, multi-purpose class — the difference is that the pre-existing complexity is now compartmentalised and, therefore, better organised [9].

Techniques to allow the quantitative measurement of cohesion can be a useful tool when restructuring code. The original definition of cohesion in [10] was difficult to assess objectively, but since then techniques that use information at the code and design levels have been developed. For example, [11] presents a model based on data slice abstraction to quantify functional cohesion, a program slice being the section of code that affects a particular program variable; a data slice is a variation on program slices. Additionally, other approaches [12] use design-level information and both association- and slice-based techniques to formally define cohesion measures.

III. LOOSE COUPLING

Coupling is concerned with the interdependence of classes (or routines), varying from *loose* to *tight*. Loose coupling is desirable and is the complement of high cohesion; the two concepts are often considered jointly when analysing

TABLE II: Common Types of Coupling, Ordered in from Most to Least Acceptable (adapted primarily from [6] but also [8] [14]).

Coupling Type	Definition
Simple data coupling	Only <i>non-structured data</i> is passed between two classes via parameter lists
Structured data coupling	Only <i>structured data</i> is passed between two classes via parameter lists
Simple object coupling	A class or routine instantiates a particular object (they are then simple object coupled)
Object-parameter coupling	<i>Object A</i> requires <i>Object B</i> to pass it an <i>Object C</i>
Control coupling	Element(s) of control are passed between one object and another, controlling how that object functions
Common coupling	Two classes both use the <i>same global data</i> (or state), with resulting behaviour impacts being unclear

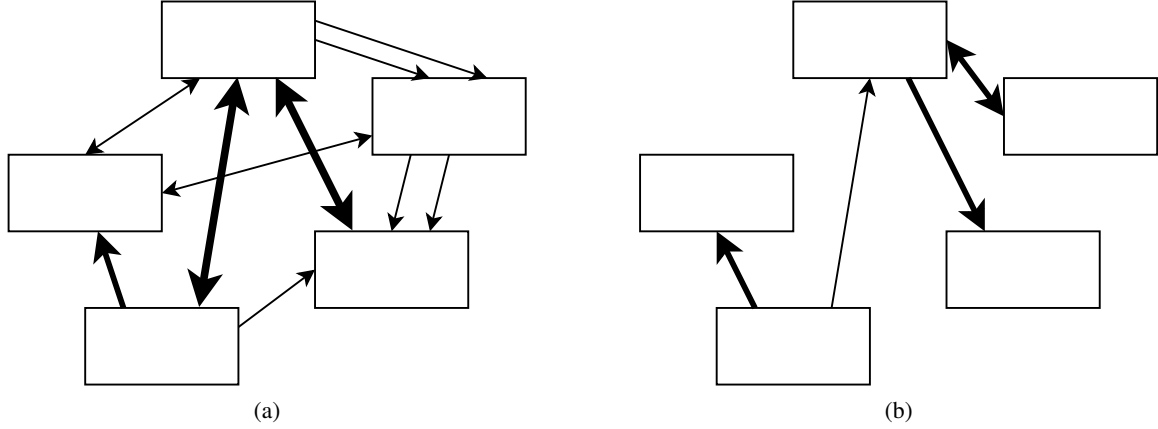


Fig. 2: Abstract visualisation of (a) a tightly coupled system and (b) a loosely coupled system. Arrow thickness shows the coupling strength (adapted from [1]).

a software’s structural complexity [13]. A class can be said to be loosely coupled if it is weakly connected to other classes, with the connections that do exist being small, explicit, direct, visible, and flexible [14]. Several kinds of coupling that are common are described in Table II.

If a class is highly coupled, a change to one class will necessitate changes to others. This reduces the modularity of the code, not to mention increasing the complexity due to many bidirectional dependencies. This concept is demonstrated in Fig. 2. In a loosely coupled set of classes, code can easily be reused as only the relevant class needs to be imported, rather than multiple interdependent classes that require each other to function correctly [1].

Early on in most programming tuition it is stressed that global data is normally to be avoided. *Common coupling* (CC) is one of the reasons that this is considered best practice. For example, [15] looks at CC in the context of Linux kernel maintainability using an analysis of definition-use patterns; with version number, the number of code lines in each kernel module increases *linearly*, whilst the instances of CC between each kernel modules grows *exponentially*.

Loose coupling also makes testing easier and more efficient. If a class is tightly coupled via implicit dependencies, or has many responsibilities, writing unit tests becomes more difficult to write and understand. If a class is hard to test then that could be an indication that the code needs to be re-factored so that its structure is improved [16]. However, the inverse may not necessarily be true. Research is inconclusive whether a test-driven approach (such as Test-Driven Development) objectively reduces coupling (or increase cohesion) [17] — recent analysis [18] suggests the reasonable assertion that “context is key”.

IV. ABSTRACTION, DATA HIDING, AND INFORMATION HIDING

Data hiding and information hiding are both heavily linked to the concepts previously discussed, with this being most apparent through the idea of *abstraction* as a technique to manage complexity [14]. Abstraction allows the

```

1 public class Point
2 {
3     public double x;
4     public double y;
5 }

```

(a)

```

1 public interface Point
2 {
3     double GetX();
4     double GetY();
5     void SetCartesian(double x, double y);
6     double GetR();
7     double GetTheta();
8     void SetPolar(double r, double theta);
9 }

```

(b)

Fig. 3: Example code for the data of a point on a Cartesian plane: (a) The data is encapsulated, but the information is not hidden. Its implementation is exposed, with it being clear that rectangular coordinates are used. (b) The information is hidden, as externally it is unknown whether the implementation within the class uses rectangular or polar coordinates (adapted from [9]).

interaction with data at different levels of complexity, with details not relevant to that given level being hidden (and therefore ignored).

Information hiding, generally accepted to have been formalised in [19], aims for details of a system that are likely to change independently to be hidden in separate classes, with only assumptions that are unlikely to change appearing in the class interfaces. For example, if *Class A* requires information that is stored in *Class B*, the information can only be obtained by calling access programs belonging to *Class B* [20]. This means that the information will be kept hidden and minor changes to a system will (ideally) be limited to the class interface; effects of change are then localised.

Data hiding is primarily distinguished from information hiding through the concept of *encapsulation*, whereby an object contains both state and behaviour together, encapsulating them into one entity [21]. Encapsulation enables data hiding, and is derived from information hiding (but is a distinct concept); both are linked to loose coupling and high cohesion, however. In general, a highly cohesive class is more likely to enable the encapsulation of data and a loosely coupled class is more likely to hide information.

The term data hiding is less commonly used in the literature. There is often the conflation of data hiding and encapsulation, such as when talking about data structures (specifically data access routines) [22]. In other cases, information hiding and encapsulation are implied but the differences are not explicitly stated [9], such as in the example in Fig. 3. With the increasing recognition of the importance of cyber security, encapsulation is relevant in this area of research, such as the concept of encapsulating internal types as a starting point for ensuring integrity and confidentiality (although current research shows that it does not guarantee it by itself) [23] and also enabling cross-platform analysis for IoT penetration testing through the use of abstract languages (such as LLVM) [24].

V. SOFTWARE ANALYSIS AND ADOPTION BY INDUSTRY

Linking together the four concepts of this report is an ongoing topic of research, and their interconnectedness is, perhaps, the most important point. Analysis of software organisation uses these concepts as a starting foundation [20]. They also allow concept analysis and module restructuring through providing a basis for assessing their quality [25].

There have also been efforts to develop models which rely on these concepts to practically assess design quality. Metrics have been developed that combine empirical and anecdotal information to quantify design properties and the quality attributes the result from them. Notable examples are the MOOD set of metrics for OO design [26] and related extensions, such as the QMOOD hierarchical model [27]. The principles can also be used to identify design flaws and construct a framework for assessing technical debt [28].

There is debate about the gap between research and practice by industry. Some authors suggest that the gap is widening, partly due to the perceived difference between theory and useful practices by industry practitioners. For example, formal models including *ad hoc* explicit state representations to apply information hiding, and consequently adding complexity, rather than considering hidden state to be the norm (for variables) [29].

VI. CONCLUSION

In this report, the concepts of high cohesion, loose coupling, data hiding, and information hiding were defined and discussed. An overview of the different types of cohesion and coupling, with their desirability (or undesirability), was given and examples visualising the basic ideas of these principles were given. The concepts of abstraction, data hiding, encapsulation, and information hiding were introduced and the overlap and distinctions between them were detailed. Research on the principles and how they can be used to increase software quality through improved design was given throughout. Nuances in conclusions from this research, such as how test-driven approaches may not necessarily reduce coupling but difficulty in testing may indicate loose coupling, were noted.

It can be confidently said that these four design principles are integral features of good software design, with peer-reviewed literature reinforcing this conclusion. However, it was noted that this is not always easy to achieve in practice and is an ongoing topic of discussion in the field.

REFERENCES

- [1] T. Lethbridge and R. Laganier, *Object-Oriented Software Engineering: Practical Software Development using UML and Java*, 2nd ed. McGraw-Hill, 2005.
- [2] A. B. Pillai, *Software Architecture with Python*. Packt Publishing Ltd, 2017.
- [3] G. J. Myers, *Reliable Software Through Composite Design*. Charter Publishers, 1975.
- [4] T. Stolarczyk, "The forgotten realm of cohesion," Accessed 28/12/2020. [Online]. Available: <https://mrpicky.dev/the-forgotten-realm-of-cohesion>
- [5] R.-G. Urma and R. Warburton, *Real-World Software Development: A Project-Driven Guide to Fundamentals in Java*. O'Reilly Media, Inc., 2020.
- [6] E. Yourdon and L. J. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1979.
- [7] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.
- [8] J. F. Dooley, *Software Development, Design and Coding*, 2nd ed. Springer, 2017.
- [9] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2009.
- [10] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.
- [11] J. M. Bieman and L. M. Ott, "Measuring Functional Cohesion," *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 644–657, 1994.
- [12] J. M. Bieman and Byung-Kyoo Kang, "Measuring Design-Level Cohesion," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 111–124, 1998.
- [13] D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko, "The Structural Complexity of Software: An Experimental Test," *IEEE Transactions on Software Engineering*, vol. 31, no. 11, pp. 982–995, 2005.
- [14] S. McConnell, *Code Complete*, 2nd ed. Microsoft Press, 2004.
- [15] L. Yu, S. R. Schach, K. Chen, and J. Offutt, "Categorization of Common Coupling and its Application to the Maintainability of the Linux Kernel," *IEEE Transactions on Software Engineering*, vol. 30, no. 10, pp. 694–706, 2004.
- [16] S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley, 2009.
- [17] D. Janzen and H. Saiedian, "Does Test-Driven Development Really Improve Software Design Quality?" *IEEE Software*, vol. 25, no. 2, pp. 77–84, 2008.
- [18] I. Karac and B. Turhan, "What Do We (Really) Know about Test-Driven Development?" *IEEE Software*, vol. 35, no. 4, pp. 81–85, 2018.
- [19] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Commun. ACM*, vol. 15, no. 12, p. 1053–1058, Dec. 1972.
- [20] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3, pp. 259–266, 1985.
- [21] The Open University, *M250 Object-oriented Java programming. Units 1-5: Getting started with objects*. The Open University, 2012.
- [22] M. Fowler, "Data Access Routines," *IEEE Software*, vol. 20, no. 6, pp. 96–98, 2003.
- [23] A. Dann, B. Hermann, and E. Bodden, "ModGuard: Identifying Integrity Confidentiality Violations in Java Modules," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.
- [24] C. Chen, Z. Zhang, S. Lee, and S. Shieh, "Penetration Testing in the IoT Age," *Computer*, vol. 51, no. 4, pp. 82–85, 2018.
- [25] P. Tonella, "Concept Analysis for Module Restructuring," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 351–363, 2001.
- [26] R. Harrison, S. J. Counsell, and R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 491–496, 1998.

- [27] J. Bansiya and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.
- [28] R. Marinescu, "Assessing Technical Debt by Identifying Design Flaws in Software Systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9:1–9:13, 2012.
- [29] D. L. Parnas, "Really Rethinking 'Formal Methods'," *Computer*, vol. 43, no. 1, pp. 28–34, 2010.

The Importance to Database Systems of Transaction Processing

Zack Costello

Student Number: 2012480

2012480@student.uwtsd.ac.uk

Module: ACCA7002

Word Count (excluding Abstract, Headers, Tables, Figures, and References): 1490

Abstract

This report is concerned with the importance to database systems of transaction processing. Essential to the concept of transactions are safety guarantees, which are described here using the ACID acronym. Each letter in the acronym is defined and the reasons for their inclusion are discussed, with practical limitations in their realisation acknowledged and associated key concepts mentioned. A fundamental technique that affects Isolation (as part of the ACID acronym), concurrency, is then explored in more detail, including common methods of concurrency control and a typical technique to achieve serialisation, locking. Throughout this report, relevant literature is referenced to reinforce key research, including both peer-reviewed academic papers and textbooks.

I. INTRODUCTION

Transaction processing is a fundamental concept for database management systems (DBMS), being essential to the processing of digital information. The term transaction processing covers the techniques that manage both the applications which process information and the actual information being processed; it is a mechanism that underpins a very large proportion of our critical resources that manage information.

Transactions are a unit of work in a database system that are logical and indivisible, allowing multiple operations (reads and writes) to be represented as a single step, simplifying the issues that arise from faults in data systems. The issues are simplified by a transaction either succeeding (a *commit*) or failing (a *rollback* or *abort*); if the transaction fails, the application simply retries. The implication of this is that it removes the issue of partial failures, which results in vastly simplified error handling (at least compared to the alternative) because an application can ignore some error scenarios and issues with concurrency as this is dealt with by the database. These are termed safety guarantees [1].

This report is concerned with discussing the importance of transaction processing to database systems. In Chapter II, the idea of safety guarantees is outlined and its relevance detailed, and Chapter III examines the issue of concurrency in database systems.

II. SAFETY GUARANTEES

A description of the safety guarantees that transactions must have are commonly described by *Atomicity*, *Consistency*, *Isolation*, and *Durability* (ACID), termed in 1983 [2]. Conversely, systems not meeting the ACID criteria can be termed *Basically Available*, *Soft state*, and *Eventual consistency* (BASE) [3]. This section discusses the four constituent parts of the ACID acronym. These are not mathematical axioms, so there is flexibility and overlap in their usage, but they are a useful organisational tool for assessing transaction processing safety.

A. Atomicity

In the context of ACID, atomicity refers to the idea that a transaction should only execute once, with all of the work being completed (committed) in that execution or none of it (the transaction would then be aborted) [4]. The transaction could be composed of multiple steps, meaning that if any one of those steps failed then it would be aborted. In the case of an abort, all of the effects of updates that were executing at the time of failure would be undone, ensuring that only transactions that are fully successful are allowed to become permanent parts of a database [5].

B. Consistency

The consistency of the database should be maintained by a transaction program; if a database, or data, is consistent at the start of a transaction, then it should be consistent at the end. More specifically, a database should be *internally* consistent, meaning that all of its integrity constraints are satisfied. These of course vary between databases, but examples include requiring all primary key values to be unique and requiring certain data values to be within a particular range [5].

However, there is debate as to whether the C in ACID actually belongs to the acronym because consistency (in the context of ACID) is an application property rather than a database property, which all of the other “letters” (AID) are. It has been suggested that it was added in order to form a workable acronym, rather than attitudes shifting on its importance and its importance (within the ACID acronym) declining over time [2].

C. Isolation

Isolation is concerned with several clients accessing a database at the same time and ensuring that these concurrently executing transactions do not interfere with each other. Formally, isolation is termed *serialisation*, (transactions) running one after the other with no overlap, and if the effect of transactions running concurrently is the same as if they were running serially, then that execution would be serialisable [6].

Isolation levels determine when the visibility of changes to a database are made and what changes are allowed to be (or become) visible to concurrent transactions. Concurrency control, discussed in Section III, is linked to isolation, with different approaches affecting whether changes are made visible to other transactions that are running concurrently [7]. In addition, serialisation can clearly have an affect on performance. It should be noted that levels of isolation may stray from the formal definition in real databases for this reason. Modern DBMSs often use a locking protocol to implement isolation.

D. Durability

If a transaction is successfully committed, it is expected that the resultant data can be safely stored and has *durability* against issues such as faults in hardware or the database crashing. This is an ideal, and achieving perfect durability is impossible; hardware and software are by nature imperfect, and “perfect storms” where several faults occur at the same time can happen (and have happened in the past). Therefore, durability is something to aim for and requires consistent effort to maintain, such as maintaining stable storage [8]. Logging and recovery are typical techniques for implementing durability.

III. CONCURRENCY

As mentioned in Section II, *isolation* is a less formal term than *serialisation* and is focused on single transactions, whilst serialisation defines correctness for transactions that are concurrent. If there are no anomalies in concurrency then a transaction is regarded as executing in isolation [9].

To clarify, concurrency in the context of database transactions is the access of data by multiple users at the same time, or at least the illusion of this happening from the users’ perspective; *transactions running in parallel*. Transactions can be run in parallel (safely) if they do not touch the same data. Issues with concurrency, sometimes known as *race conditions*, only occur when the same data is/are modified by two transactions, or when data is read by one transaction whilst being modified concurrently (in parallel) by another [8].

A. Enforcement of Concurrency Control

To enforce currency control there are three broad techniques [7]:

- 1) Optimistic Concurrency Control (OCC): OCC is based on whether the combined execution of concurrent read/write operations is serialisable. If multiple transactions conflict then one of them is aborted.
- 2) Multi-Version Concurrency Control (MVCC): MVCC allows multiple versions of a record to be present in a database by timestamping them. This guarantees that the database is, at a minimum, consistent at at least one point in the past.

- 3) Pessimistic Concurrency Control (PCC): PCC can be based on locking or non-locking methods, differing by shared resource access/grant management. Lock-based methods require locks to be maintained on database records, preventing them from being modified or, until the transaction lock has been released, assessed whilst they are being modified. Methods that are non-locking allow read/write operation lists and restrict execution based on the unfinished transaction schedule.

B. Locking

Whilst most components of a DBMS can be split into independent parts, the transactional storage manager is one piece that is typically large and typically includes four components that are difficult to isolate from each other [10]:

- To implement currency control, a lock manager.
- To implement recovery, a log manager.
- To stage database input/outputs, a buffer pool.
- To organise data on a disk, access methods.

To achieve serialisation, the most common mechanism used is locking. The concept is based on the idea that when a transaction is using data it reserves access to that data, with this reservation being termed a lock (either read or write). Locks aim to avoid transactions interfering with each other, with conflicting locks being used to synchronise operation execution order [11].

Before data is read, a read lock is set by a transaction. Alternatively, before data is written, a write lock is set by a transaction. There are conflicts that can occur; write locks conflict with both write locks and read locks, whilst read locks only conflict with write locks. If there is no conflicting lock on a data item, then a transaction is allowed to obtain a lock.

Locking can affect transaction performance, such as when multiple transactions need to set conflicting locks; in general, the more concurrently-running transactions, the more delays will occur. The mechanisms of locking are beyond the scope of this report, but it should be noted that the performance impact of locking is affected by the design of transactions, database layouts, and the distributions of both transactions and databases. A very widely-used serialisation algorithm is two-phase locking (2PL), and should be noted here due to its influence [5].

IV. CONCLUSION

In this report, the importance of transaction processing in the context of database systems has been outlined and discussed. The concept of safety guarantees was introduced, along with the ACID acronym. One important area of transactions, concurrency, was defined. Three common methods of concurrency enforcement were described and an important method of achieving serialisation, locking, was examined.

Transactions are not necessary for all applications, and they can negatively affect performance and/or the availability of data — concurrency is one consideration which influences these factors. However, they simplify many of the issues that database designers have to contend with. This simplification of error handling explains its adoption and continued popularity; applications accessing a given database are presented with a simplified programming model, providing layers of abstraction that allow other developers to focus on their constituent part of the overall architecture.

REFERENCES

- [1] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufman Publishers, Inc., 1992.
- [2] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, p. 287–317, Dec. 1983.
- [3] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services," in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 78–91.
- [4] Microsoft, "ACID properties," Accessed 03/01/2021. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/cosssdk/acid-properties>
- [5] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, 2nd ed. Morgan Kaufman Publishers, Inc., 2009.
- [6] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database System*. Addison-Wesley, 1983.

- [7] A. Petrov, *Database Internals: A Deep-Dive into How Distributed Data Systems Work*. O'Reilly Media, Inc., 2019.
- [8] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, Inc., 2017.
- [9] P. P. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, vol. 13, pp. 185–, 06 1981.
- [10] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, "Architecture of a Database System," *Foundations and Trends in Databases*, vol. 1, no. 2, pp. 141–259, 2007.
- [11] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd ed. McGraw-Hill Higher Education, 2003.