

LEVEL 7 ASSESSMENT SPECIFICATION

Student name:	Owen Skilton	Student ID number:	2013056
Programme:	MSc Software Engineering (Web Development)		
Module:	Programming and Software Testing Database Systems		
Module code:	ACCA7000 ACCA7002	Contribution to Overall Module Assessment (%):	50% 50%
Lecturer:	Tim Bashford Beverley Holland	Internal Verifier:	Beverley Holland
Assignment Title:		Word count (or equivalent):	6000
Submission deadline:	15/01/2021	Return date of provisional marks & written feedback:	
Submission method:	All written assessments, where practical and possible, must be submitted via Turnitin unless otherwise instructed by the Lecturer. (Please DO NOT put this assessment specification into Turnitin or it will match many similarities with other students' submissions.) Alternative submission method (if applicable): Late submission of the assessment will result in a late penalty mark. Penalties for late submission: Up to one week late, maximum mark of 50%. Over one week late, 0%. Only the Extenuating Circumstances Panel may approve a change to submission dates.		
Academic honesty / referencing:	Academic honesty is required. In the main body of your submission you must give credit to authors on whose research and ideas your work is based. Append to your submission a reference list that indicates the books, articles, etc. that you have used, cited or quoted in order to complete this assessment.		

Table of Contents

1. Programming.....	3
1.1 Reflective report	3
1.1.1 Reflective	3
1.2 Technical report	3
1.2.1 Introduction	3
1.2.2 Definitions	4
1.2.3 Summary	6
1.2.4 References	6
2. Databases.....	7
2.1 Reflective report	7
2.1.1 Reflective	7
2.2 Technical report	8
2.2.1 Introduction	8
2.2.2 ACID	8
2.2.3 Concurrency control	9
2.2.4 Summary	10
2.2.5 References	10

1. Programming

1.1 Reflective report

1.1.1 Reflective:

A key design decision to note is that all user input is run through a data validation function to ensure it meets both the format check for the SQL table as well as the data type needed to interact correctly with the prepared statements when they are executed. This technique however is not very progressive as it requires updates to the validation function if a table or column is ever added to the database in the future, unless the code is edited to create separate function validation again such as the ones performed with dates, telephone numbers and registration plates.

The instances where prepared statements were not used such as the view all function, the validation for the user input would only allow 6 string values to enter the query regardless of user entry. The same can not be said however for one function, the update a record method. This method passes its valid entries into a string builder to for the SQL condition for the query. This is a fundamental flaw with the function and would require a complete redesign in order to avoid SQL injection. This also ties into the unit test coverage ability for the SQL queries. A few are for dependent meaning they are a bit too complex to test for individual unit tests so a redesign of the over architecture would allow for more unit test coverage over all and would avoid injection attacks at the same time.

1.2 Technical report

1.2.1 Introduction:

Within the modern day of software development, it is crucial that working practices are able to maintain pace with ever adapting demand for functionality and maintenance of software being created. With deadlines needing to be met and more and more complex functionality being required of services as well as customer demands it is vital that any software being produced is able to possess some form or aspect of reusability, ease of maintainability and simplicity for others to understand.

Upon release of a software product, maintenance and evolution is required to be continuous throughout the products life cycle. High dependency between modules within the software will increase the probability that changes to one module will have inverse effects and introduce failures within the future. [1] In order to avoid these future failures and increase the ease of maintenance and reusability of the software's architecture, it is crucial to implement logical methods that can define dependencies and the relationships between software modules to allow for use within the future.

With those key features in mind, the following report will contain information that pertains to both attributes associated with the dependency and relatability of software modules, coupling and cohesion respectively, as well as the methods that ensure that these dependency attributes are maintained through out the development cycle within software development.

1.2.2 Definitions

Cohesion

Cohesion is defined as an internal software attribute that represents the degree to which the components within a software module are related together. Members within a class that are highly cohesive possess basic functionality and are difficult to decompose down into a more basic form. The inverse is true for low cohesion, methods that require a complex set of operations to be performed within it and are easily broken down into separate components are undesirable as they limit the usability as well as testability of the entire method. Within the scope of object oriented programming, cohesion is a highly desirable trait for components as external attributes such as reusability, maintainability and reliability can be improved simply by making components more cohesive. [2] There are a multitude of ways to measure the cohesiveness of the components within a class such as the one hypothesized by Heung S.C et al [3] where by they take into account the nature of such methods within a class that possess different attributes such as access methods, constructors and destructors by attributing values to these 'special methods' that don't decrease the cohesiveness. This differs from conventional cohesion calculations that use the interaction between members as the dominant factor for their calculations.

In simplified terms, cohesion is the strength that the relationships between different classes possess as well as the simplicity of actions these methods perform with complexity kept to a minimum, meaning less complex methods are more desirable as the software will possess traits that are advantageous such as reliability, reusability where as low cohesion possesses traits such as difficulty to test, the ability to understand the functionality as well as to maintain it. Cohesion often correlates to a separate degree of measurement in coupling.

Coupling

Coupling essentially is the contrast to cohesion, it's a measure of the degree of interaction between 2 separate software modules where it is desirable to achieve the same core attributes of software such as understandability, maintainability and reusability. [4] Loose coupling relates to a low degree of dependency whereby the modules possess low complex methods with desirable traits, similar to high cohesion. The inverse is again true for strong coupling which relates to highly dependent software modules with a large amount of complexity and low reusability. It has been shown that modules that possess strong coupling tend to be prone to faults when exposed to modifications and future maintenance. [5]

Where coupling differs from cohesion however is that there are different types of coupling where the degree of dependency differs depending on the type of couple that is formed. In order of dependency rank, the different types of coupling are as follows: [6]

- Content coupling:
 - o 1 module accesses and alters the data or logic of the other. An example of this would be a branch switch whereby you inherit and alter all the data from another module.
- Common coupling:
 - o Modules are common if they refer to the same global variable
- Control coupling:

- Modules are control coupled when one passes a variable to another that is then used to control the internal logic of the other. An example of this would be a comparison function that is reliant on the data of the other to function.
- Stamp coupling:
 - When 2 modules pass data through a parameter that is a record. This could be redundant if the module receiving only needs one field of the record which leads to Data coupling
- Data coupling:
 - Modules are data coupled when they pass data between each other that is a singular value such as an integer variable.

When it comes to calculating the value for coupling between 2 modules, depending on the type of couple will add weight to the value for the dependency for the system.

Data hiding

Data hiding, also known as data encapsulation, within software development is specific technique that is used within object orientated programming to hide internal objects within a class. This technique ensures that data access is exclusive to the related class members meaning the integrity of the object is maintained and protects against any unintended changes. This means that all functionality related to a specific object that is being interacted by the user will come through specific methods which then restricts the set of operations that can be performed on said class and by extension the data within it, maintaining integrity. Data hiding is an important method to utilise as by incorporating it into software it reduces the complexity of the system which also reduces the dependencies between interacting components. [7]

Data hiding is a useful extension to the metrics of cohesion and coupling as the simple incorporation of it can enable the parameters of the software to be closer to a desired product allowing for better understanding and reusability. By creating specific and simple methods that interact with objects within the class, the user maintains intractability and functionality while also being restrictive in the application of retrieval to the remaining data within said class. The emphasis on the simplicity of the function increase the cohesive nature of the modules ensuring easy maintainability and reusability in the future.

Information hiding

Although the terms are used interchangeably depending on the resources used, information hiding differs from the concept first proposed by Zilles in 1973 relating to data hiding. [7] Encapsulation may refer to the combining of data with the methods that are used to manipulate said data, information hiding can be classed as the design principle behind it that ensures the separation of modules within a system that are most prone to alterations in the future while not compromising the overall functionality of the system. The term information hiding was first coined by Parnas in 1971 [8] who focused on ensuring that the external user interface functionality was separated from the overall system functionality to ensure that any alterations made to methods pertaining to data interaction ensured that the user experience is not compromised. The splitting of these methods has the effect of reducing the complexity of the system as well as understandability.

Encapsulation is most commonly described as a mechanism that extends from information hiding within object orientated programming which allows for the functionality of information hiding. This

means it is possible to have data hiding techniques employed while not abiding by the information hiding definition. [9]

To simplify, information hiding is the principle that not all the information in regard to a class is able to be altered and interacted with in order to ensure code functionality and crucial aspects remain consistent. This then has an effect on our key principles mentioned in cohesion and coupling whereby the software produced increases in flexibility and maintainability. Encapsulation is the implementation behind the idea of information hiding by ensuring data is wrapped within classes and is only interacted upon the use of certified methods or functions this ensures that the main principle of information hiding in integrity is maintained.

1.2.3 Summary

To summarise, information hiding is a vital component of software development ensuring that data integrity is maintained for code based functionality. Information hiding is able to be maintained by utilising methods such as data hiding/encapsulation which contains data to a class in terms of objects allowing for restricted interactions. This consequently has the effect on the 2 parameters used to define how dependent the methods used within the software are in coupling and cohesion, allowing for less dependent methods to be more understandable, maintainable and reusable.

1.2.4 References

- (1) R. D. Banker et al. Software complexity and maintenance costs. Communications of the ACM, 19993, 36(11), pp.81+.
- (2) J. Zhao, B. Xu. Measuring Aspect Cohesion, 2004. [Online] Available from: https://link.springer.com/chapter/10.1007/978-3-540-24721-0_4 [Accessed 11 Dec 2021]
- (3) H. S. Chae, Y. R. Kwon, D. H. Bae. A cohesion measure for object-oriented classes. Software: Practice and experience, 2000, 30(12), pp.1405-1431.
- (4) W. P. Stevens, G. J. Myers, L. L. Constantine. Structured design. IBM Systems Journal, 1974, 13(2), pp.115-139.
- (5) R. W. Shelby, V. R. Basili. Analysing Error-Prone System Structure. IEEE Transaction on software engineering, 1991, 17(2), pp.141-152.
- (6) L. Yu, S. Ramaswamy. Examining the relationships between software coupling and software performance: a cross-platform experiment. Journal of computing and information technology, 2011, 19(1), pp.1-10.
- (7) S. N. Ziles. Procedural encapsulation: a linguistic protection technique. *In* Proceeding of ACM SIGPLAN-SIGOPS interface meeting on Programming languages-operating systems, pp. 142-146.
- (8) D. L Parnas. A paradigm for software module specification with examples, 1971.
- (9) A. T. Cohen. Data abstraction, data encapsulation and object-oriented-programming. ACM SIGPLAN Notices, 1984, 19(1), pp.31-35

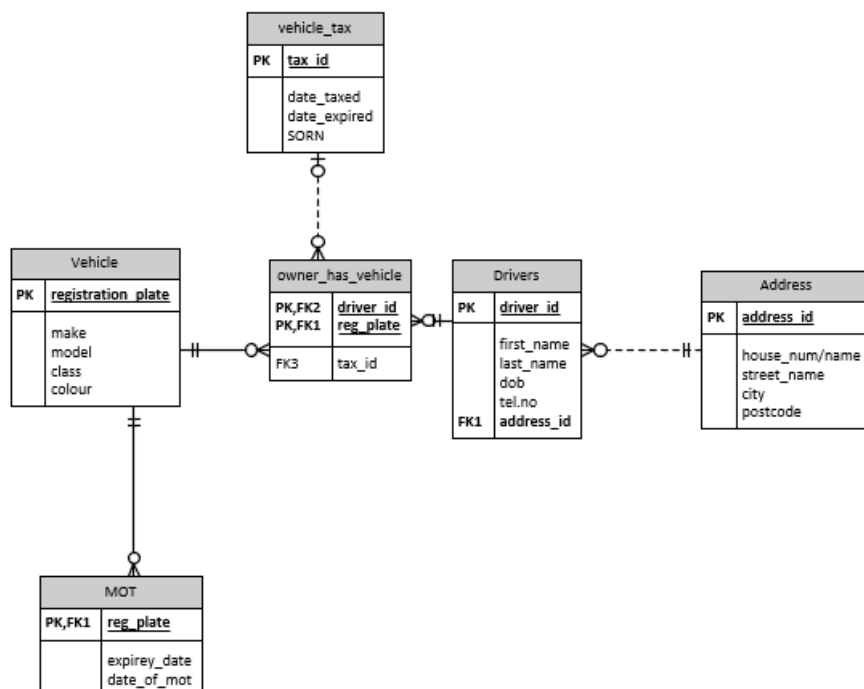
2. Databases

2.1 Reflective report

2.1.1 Reflective:

My entire database architecture is based around auto key generation within certain tables that are then inherited by other records to ensure 3rd factor normalisation is maintained. For example, driver ID, address ID, tax ID and vehicle ID are inherited by the rest of the tables whose keys are auto generated upon a submission of a record into each table starting from 1. Tables such as owner_has_vehicle mean that drivers can own multiple vehicles and multiple drivers can live at multiple addresses thanks to how address ID keys are inherited during formation of a driver record. One downside to the generation however is that they currently can only increment from 1 by 1 each record submission. It would be convenient if say driver record had a key format of DR0001 and address had AD0001, to make reading of records and searching by IDs more user friendly.

In my opinion the most important database transaction that takes place is the view driver record. Whether this is the individual record or a whole table, a majority of the test within the programming portion of this assignment rely on the ability to view records in a specific format in order to pass without failure. A downside of how the view architecture functions is that it only allows for one parameter to be inputted within the filter condition. If a user wished to search for a specific city for example, multiple amounts of records would be returned, and it would be difficult to extract the specific data the user wishes to access with no ability to filter any further. For future development a function would need to be created to facilitate multiple filter conditions. Below is the ERD diagram used for the design:



2.2 Technical report

2.2.1 Introduction

With modern day data transactions consisting of terabytes upon terabytes of data being passed, manipulated and viewed regularly, it is crucial that these transactions and architectures behind these databases are able to scale with times and ensure that seamless transactions are implemented and the users experience is kept to a high standard. To ensure transactional and data integrity, it is important to focus on the properties behind the queries being submitted. There are multiple ways to measure the level of security that a transaction possess, one way is known as ACID. ACID focuses on how the transactions being submitted interact as well as ensuring that the number of problems that arise from execution is kept to a minimum as well as the number of errors to potentially arise. Even though it has been noted that for SQL-based relational database management systems that incorporating ACID properties hampers scalability as well as flexibility in the handling of big data efficiently [1], the opposite is true for NoSQL based databases that rely on schema less architecture saving time over the traditional method as you allow for different structures to be created for each record.

Even if the transaction properties are accepted based as well as following the guidelines set out by ACID, it does not ensure that data validation and transaction success is guaranteed within a database management system. It is important to establish some form of procedure for multiple initiated transactions; do not cause conflictions with the data they are interacting with during the execute phase. This issue can be resolved by establishing some form of concurrency control which will be discussed later. The procedure ensures that the user experience, depending on application, will not be impacted by the number of transactions being executed to the management system.

2.2.2 ACID

To ensure transaction and data integrity is maintained during execution, the properties of the transactions being submitted are able to be analysed and defined to ensure that issues do not arise upon execution. With relation to transactions within a database, a possible measure meant for transaction is ACID. there are 4 key principles that are needed to be understood for efficient transactions known as the ACID test. [2] The principles are as follows:

- Atomicity:
 - Atomic transaction is defined as the execution of a transaction that occurs all at once rather than in separate sections. It is imperative that the transaction is performed all at once if it involves more than one query. For example, when moving funds from one account into another, the addition of funds into one account needs to coincide with the subtraction of the other to ensure that there is no potential failure of one of them.
- Consistency:
 - The 2nd property of the ACID test is consistency. After a transaction is complete within a database, a consistent transaction is defined as one where all the internal integrity constraints within the database are satisfied. For example, all primary keys must be ensured to be unique as well as data ranges and formatting plus referential integrity.
- Isolation:

- Isolation is defined as if the transactions were executed as one would achieve the same state / result as if the transactions were executed serially I.E one after another. Isolation runs concurrently with atomicity and work together to ensure the integrity of the database is maintained when both conditions are met.
- Durability:
 - The final transaction definition to satisfy the ACID test is durability. Durability is defined as upon completion of a transaction, the location at which this updated data is stored is known as stable storage. Stable storage means that the data stored within them is able to survive and event such as power failure. In essence its non-volatile data that is stored with devices such as magnetic disks and solid states.

A clear advantage to utilising an ACID based approach to transactional validation is that it simplifies the job of the concurrency control. Providing the ACID properties are met then the role of the concurrency control method is that it simply has to ensure that the executability of said transaction that occurs concurrently is a lot more linear. This allows for minimized conflicts and quicker transactional completions. A disadvantage however is that databases transactions that rely on ACID properties to perform efficiently in a scaled up system suffer immensely. "The traditional transactional models are problematic when loads need to be spread across a large number of components." This was mentioned by Dan Pritchett [3] in regard to the difficulty of cross module transactions that abide by a standard definition of ACID. It is important to decouple the operations and preform them in turn to improve availability at the cost of consistency. A recommended substitute for ACID is a system known as BASE which performs more efficiently at this scaled up function.

2.2.3 Concurrency control

With both public and private databases being accessed by multiple administrators and user simultaneously, it is vital that any data interaction, whether it be rewriting or manipulating, does not get conflicted across multiple user based transactions. Concurrency control is a method to ensure that the database management system being utilised ensures that the transactional order of queries is consistent across users [4]. For example, if 2 users are working within the same management system and one attempts to reference a data cell while the other attempts to edit that data, many potential conflicts and thrown errors may arise on both users' end. Concurrency helps eliminate this issue by executing the transactions simultaneously eliminating the possibility of the 2nd transaction inline being thrown an error, it effectively orders the transactions and prevents the other beginning until the other is completed, this prevents any issues arising from simultaneous executes. The order of transaction may be determined by the protocol selected by the concurrency control. Concurrency is a crucial aspect of any modern data management system as it ensures data integrity is maintained for any form of user interaction.

When introducing concurrency control into a database management system it is common that the properties for the transactions such as ACID, overlap with method for control. Even though multiple different transactions can be performed simultaneously, the resulting action is as if they are preformed one after another ensures that the properties of ACID are able to be maintained. An example method of this form of concurrency control is the lock method. [5] The lock method functions by restricting access to the data being accessed concurrently until the completion of the first transaction, ensuring that no conflicts arise during other simultaneous transactions. Other methods as proposed by Dong et al known as deterministic concurrency control (DOCC) follow the method of if 2 transactions are initialized simulations for a form of record access, transaction 2 is unable to perform validation until after the first transaction commit has been completed. [6] A common method is Time stamp based protocols, put simply, it is defined as executing the

transactions in the order that they were submitted to the database. This however does not solve all problems but for simplistic read and write methods is more than appropriate.

2.2.4 Summary

The combination of both ACID properties for transactions within a database as well as the need for some form of concurrency control to ensure transaction integrity allows for a multitude of advantageous within a databases architecture. For one it ensures to a reasonable degree of certainty that integrity and safety of the data is maintained. The chance of conflicting transactions is essential eliminated. ACID is a valuable tool for data security however it is not the definitive answer. Depending on the application of the system other methods for determining transaction properties may be utilised. For a simplistic system that does not possess multifunctionality across several modules, an ACID based approach is more than sufficient for the databases needs. However, if a scaled approach is need with multi-communicational databases, a new transactional property procedure may need to be investigated for efficiency sake, especially if the number of transactions becomes excessive and performance is a priority. Another is that the future proofing of a databases architecture is ensured, providing that scalability is not excessive. There should be a moderate amount of success with the future proof of especially internal systems and dashboards. Minor changes may need to be made to the underlying tables, but the transaction ability will unlikely be compromised.

Concurrency runs alongside the databases transaction properties and exists regardless of the method selected. Concurrency is an all encapsulating term for the procedure used for transaction execution and depending on the property type used, the concurrency method may need to be edited. A multitude of protocols exist for different applications, but most follow the pattern of serialized transaction execution.

2.2.5 References

- (1) Z. Aftab, W. Iqbal, K. M. Almustafa, F. Bukhari, M. Abdullah. Automatic NoSQL to Relational Database Transformation with Dynamic Schema Mapping. Scientific Programming, 2020, pp. 13.
- (2) P. A. Bernstein, E. Newcomer. Principles of transaction processing, Morgan Kauffman, 2009, pp.10-15
- (3) D. Pritchett. Base: An acid alternative. Queue, 2008, 6(3), pp.48-55
- (4) P. A. Bernstein, N . Goodman. Multiversion concurrency control – theory and algorithms. ACM Transactions on Database Systems, 1983, 8(4), pp.465-483
- (5) T. Kuduo, M. Ishino, K. Saotome, N, Kataoka. A proposal of Transaction Processing Method for MongoDB. KES, 2016, pp. 801-810.
- (6) Z. Y Dong, C. Z. Tang, J. C. Wang, Z. G. Wang, H. B. Chen, B. Y. Zang. Optimistic Transaction Processing in Deterministic Database. Journal of Computer Science and Technology, 2020, 35, pp.382-394.