Document database

By Ahmad al-khateeb

Atypon - Wiley

Amman, Jordan

February – 2022

CONTENTS

1 Introduction

1.1 Overview.

A document database (also known as a document-oriented database or a document store) is a database that stores information in documents. A document database is a type of nonrelational database that is designed to store and query data as JSON-like documents. Document databases make it easier for developers to store and query data in a database by using the same document-model format they use in their application code. The flexible, semistructured, and hierarchical nature of documents and document databases allows them to evolve with applications' needs. The document model works well with use cases such as catalogs, user profiles, and content management systems where each document is unique and evolves over time. Document databases enable flexible indexing, powerful ad hoc queries, and analytics over collections of documents.

1.2 Aims and Objectives.

The aim of this project is to develop a document database with the required points that will be discussed later, while keeping in mind that the development should follow the Object Oriented Programming approach, Design Patterns, Clean Code, SOLID programming Principles, the Effective Java development rules and DevOps practices. The rules listed above are the core principles of this project, they will help us achieve the best software performance and scalability so we will be going into them into details in the later chapters.

1.3 Design Requirements.

- How you implemented the DB.
- The data structures used.
- Multithreading and locks.
- Scalability/ Consistency issues in the DB.
- Security issues in the DB.
- The protocol you implemented between the client and the server.
- Defending your code against the Clean Code principles (Uncle Bob).
- Defending your code against “Effective Java” Items (Jushua Bloch).
- Defending your code against the SOLID principles.
- Design Patterns.
- DevOps practices.

1.4 Report Structure.

Within the First Chapter of this documentation, I gave a brief overview of the project to be submitted, in addition to explaining the aims and objectives of the project, and the design requirements that must be meet when developing this project.

On the other hand, the Second Chapter will discuss the implementation of the database in the development of this project.

The third chapter will discuss the Data Structures used when designing this project, followed by the fourth chapter discussing the Multithreading and locks.

Chapter five will discuss Scalability/ Consistency issues in the DB followed by Chapter six discussing Security issues in the DB.

Chapter Seven the protocol you implemented between the client and the server.

Chapter eight the Clean Code Principles used followed by the ninth chapter discussing the Effective Java Code Principles.

Chapter ten the SOLID principles used followed by the Eleventh chapter will discuss the Design Patterns used in the development of this project.

And for Chapter twelve, the last chapter in this documentation we will discuss the DevOps practices that used with this project.

2 THE IMPLEMENTATION OF THE DATABASE

2.1 Introduction

My document database is a simple database that stores the json object in text files and it has an admin and a default admin

The DB is multi-threaded and supports all CRUD (create/read/update/delete) operations. no unauthorized users can access the database as role-based access control (RBAC) approach is applied to restrict system access to authorized users only. And as a database I created a Library database.

Multiple users with different roles and responsibilities can use the system at the same time and preform CRUD operations. the dataset will be stored on the main database file and for fast read it will read from a LRUCashe.

The system will interact with 2 databases, the document DB and the Users database, to clarify the context and the boundaries of the system and what is the relationship of the system and the entities I have designed the following context diagram

Indexing is the process of associating a key with the location of a corresponding data record. There are many indexing data structures used in NoSQL databases. We will briefly discuss some of the more common methods; namely, B-Tree indexing, T-Tree indexing, and O2-Tree indexing.

Every admin has his own database and his own users and can do the following:

1-add/remove database

2-add/remove schema

3-add/remove document

4-add/remove users

5-add/remove index

6-add/remove index

I apologize for that bad implementation

TODO replica and read and observer

3 THE DATA STRUCTURES USED

3.1 Introduction

Data structure is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. In this chapter I'll be going through the data structures that I used when coding this project.

3.2 Primitive Data Structures

Primitive Data Structures are the basic building blocks of any programming language, some call them data types they have been used widely in this project differing from Integers, Strings, Booleans and so on. Because of their wide usage I will not be giving an example on were they have been used.

3.3 Non-Primitive Data Structures

N.P.D.S. have two subtypes Liner Data Structures and Non-Linear Data Structure, Non-Linear Data structure

I used three types of them.

1. Map

    a. Ex. Used when storing the cache.

```
private HashMap<K, Node<K, V>> cache;
```

    b. Ex. Used for storing the document.

```
HashMap<Integer,String> documents=new HashMap<>();
```

2. ArrayList

    a. Ex. Used for storing the databases.

```
private ArrayList<String> dataBasesList = new ArrayList<>();
```

3. BTree

    a. Ex. Used for search on documents.

```
public class BTree1<Key extends Comparable<Key>, Value>
```

In Conclusion using the appropriate Data Structures will guarantee better performance when running the program. Selecting the proper Data Structure is left off to the developer after he knows the tradeoffs of each data structure.

4 MULTITHREADING AND LOCKS

4.1 Introduction

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing.

Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks.

- I didn’t do locks for now maybe after I finish I will do

example of thread pool

```
private static ArrayList<ClientHandler>Students=new ArrayList<>();

private static ExecutorService pool = Executors.newCachedThreadPool();

public static void main(String[] args) throws IOException {


        ServerSocket serverSocket =new ServerSocket( port: 8000);
        while(true){
            Socket socket = serverSocket.accept();
            System.out.println("the connection is done");
            ClientHandler studentThread = new ClientHandler(socket);
            Students.add(studentThread);
            pool.execute(studentThread);
        }
}
```

example of synchronized:

```
public boolean deleteSchema(String path,String schemaName) {
    synchronized (this) {
        try {
            File file = new File( pathname: path + "/" + schemaName);
            if (file.exists()) {
                jsonSchema.remove(schemaName);
                FileUtils.deleteDirectory(file);
                return true;
            }
```

5 SCALABILITY/ CONSISTENCY ISSUES IN THE DB

Docker Swarm:

Docker Swarm Explained: A Docker Swarm is a group of either physical or virtual machines that are running the Docker application and that have been configured to join together in a cluster. Once a group of machines have been clustered together, you can still run the Docker commands that you're used to, but they will now be carried out by the machines in your cluster. The activities of the cluster are controlled by a swarm manager, and machines that have joined the cluster are referred to as nodes.

**I didn’t try it yet but I will I just read about its implementation .

A reference to read https://docs.docker.com/engine/swarm/

6 SECURITY ISSUES IN THE DB

There is weak security in my database just that check for users if they are exist or not. It doesn't let anyone that is not in the database to see other databases.

For the password it saved as String not encrypted. the user password by getting the hash value for password , Generate Salt value

Like SHA256 Hashing Technique: SHA is the Secure Hash Algorithm. It uses a cryptographic function that takes up the 32-bit plain-text password and converts it into a fixed size 256-bit hash value.

SHA512 MD5 Hashing Technique: SHA512 uses a cryptographic function that takes up the 64-bit plain-text password and converts it into a fixed size 512-bit hash value.

7 THE PROTOCOL BETWEEN THE CLIENT AND THE SERVER.

I used socket server

Definition:

A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

8 THE CLEAN CODE PRINCIPLES (UNCLE BOB)

8.1 Introduction

In this chapter we will be going through all the code smells that are mentioned in Robert Martin's Clean Code book, many of the code smells are already discussed in Martin Fowler Refactoring book with some newly added smells, you as the reader could refer to any of the smells discussed here and read more about it in the mentioned books.

8.2 Code Smells

8.2.1 COMMENTS

1. Inappropriate Information.

    a. Definition: It is inappropriate for a comment to hold information better held in a different kind of system such as your source code control system, your issue tracking system, or any other record-keeping system.

    b. Project Use: This point has been used extensively when coding the project since it's used everywhere there is no need to specify where it has been used.

2. Obsolete Comment.

    a. Definition: A comment that has gotten old, irrelevant, and incorrect is obsolete. Comments get old quickly. It is best not to write a comment that will become obsolete. b. Project Use: This point has been used extensively when coding the project since it's used everywhere there is no need to specify where it has been used.

3. Redundant Comment.

a. Definition: A comment is redundant if it describes something that adequately describes itself.

b. Project Use: This point has been used extensively when coding the project since it's used everywhere there is no need to specify where it has been used.

4. Poorly Written Comment.

    a. Definition: A comment worth writing is worth writing well. If you are going to write a comment, take the time to make sure it is the best comment you can write.

    b. Project Use: This point has been used extensively when coding the project since it's used everywhere there is no need to specify where it has been used.

5. Commented-Out Code.

a. Definition: Code that has been commented out by the developer and never used while forgetting to delete it or keeping it for other developers to look at it.

b. Project Use: No code that has been commented out is left behind.

8.2.2 ENVIRONMENT

1. Build Requires more than one step.

a. Definition: Building a project should be a single trivial operation. You should not have to check many little pieces out from source code control You should not have to search near and far for all the various little extra JARs, XML files, and other artifacts that the system requires.

b. Project Use: Since this is a Maven Project, the developer will not need to add any dependences just import it as a maven project and run “mvn clean install” in the terminal.

2. Tests Require more than one step.

a. Definition: You should be able to run all the unit tests with just one command. In the best case you can run all the tests by clicking on one button in your IDE.

b. Project Use: I write some test cases.

8.2.3 FUNCTIONS

1. Too Many Arguments

a. Definition: Functions should have a small number of arguments. No argument is best, followed by one, two, and three.

b. Project use: All the functions in the project are minimized to take no arguments or three arguments in the most cases.

2. Output Arguments

a. Definition: If your function must change the state of something, have it change the state of the object it is called on.

b. its generally have been used in multiple places around the project to eliminate the code smell.

3. Flag Arguments

a. Definition: Passing a Boolean into a function is a truly terrible practice. It immediately complicates the signature of the method.

b. Project Use: Refer to Figure 2 for a demonstration on where this rule has been used, its generally have been used in multiple places around the project to eliminate the code smell but the figure will give an example. Figure 10 : Flag Argument Code Smell

4. Dead Function

a. Definition: Methods that are never called should be discarded. Keeping dead code around is wasteful.

b. Project Use: All the functions in the project that are not being used are deleted

8.2.4 GENERAL

1. Multiple Languages in One Source File

a. Definition The ideal is for a source file to contain one, and only one, language:

b. Project Use: No source file in the project contains more than one programming language.

2. Obvious Behavior Is Unimplemented

a. Definition: any function or class should implement the behaviors that another programmer could reasonably expect

b. Project Use: All the functions in the project implement the behaviors that the programmer is expected to use.

3. Incorrect Behavior at the Boundaries

a. Definition Every boundary condition, every corner case, every quirk and exception represent something that can confound an elegant and intuitive algorithm

b. Project Use: All the boundary conditions I could have reasoned have been dealt with edge cases or a try catch block, refer to Figure 3 below for an example that has been used in the code

```
    try {
        Files.delete(file.toPath());
        newFile.renameTo(file);
    } catch (Exception e){
        System.out.println("not found");;
    }
}
```

4. Duplication

    a. Definition: Every time you see duplication in the code, it represents a missed opportunity for abstraction, and you should always refer to the DRY rule don't repeat yourself.

    b. Project Use: The code has been refactored to follow the DRY principle, while adding abstraction and factory pattern to reuse the code that would be used in another class, this point has been discussed into detail in the previous chapters.

5. Code at Wrong Level of Abstraction

    a. Definition: constants, variables, or utility functions that pertain only to the detailed implementation should not be present in the base class. The base class should know nothing about them.

    b. Project Use: This rule along with rule four have been discussed in the design patterns section of this documentation.

6. Base Classes' Depending on Their Derivatives

    a. Definition: partitioning concepts into base and derivative classes is so that the higher-level base class concepts can be independent of the lower level derivative class concepts

    b. Project Use: Base Classes are never dependent on their derivatives in this project, and I didn't have to deal with and case that this rule is violated.

7. Too Much Information

    a. Definition: A well-defined interface does not offer very many functions to depend upon, so coupling is low. A poorly defined interface provides lots of functions that you must call, so coupling is high.

    b. Project Use: I tried to minimum the functions but it's the admin class.

8. Dead Code

    a. Definition: Dead code is code that isn't executed.

    b. Project Use: No dead code has been left off in the project

9. Vertical Separation

a. Definition: Variables and function should be defined close to where they are used. Local variables should be declared just above their first usage and should have a small vertical scope.

b. Project Use: Variables and functions that are meant to be used in the same scope have followed this rule.

10. Inconsistency

a. Definition: If you do something a certain way, do all similar things in the same way.

b. Project Use: no I didn't use.

11. Clutter

a. Definition: Variables that aren't used, functions that are never called, comments that add no information, and so forth, all these things are clutter and should be removed. Keep your source files clean, well-organized, and free of clutter.

b. Project Use: No Clutter code has been left off in the project.

12. Artificial Coupling

a. Definition: Things that don't depend upon each other should not be artificially coupled. For example, general enums should not be contained within more specific classes because this forces the whole application to know about these more specific classes.

b. Project Use: While programming this project I tried to find where there could be coupling in the classes' and decouple them, refer to the Figure below for an example where I decoupled the Enums into a separate class.

```
public enum Role {
    ADMINISTRATOR,USER;
}
```

13. Selector Arguments

    a. Definition: Selector arguments combines many functions into one. Selector arguments are just a lazy way to avoid splitting a large function into several smaller functions

    b. Project Use: When designing the project, I didn't use any selector arguments for the sake of avoiding the creation of multipurpose functions.

14. Function Names Should Say What They Do

    a. Definition: If you must look at the implementation or documentation of the function to know what it does, then you should work to find a better name or rearrange the functionality so that it can be placed in functions with better names.

    b. Project Use: This point has been used extensively when coding the project since it's used everywhere there is no need to specify where it has been used.

15. Use Explanatory Variables

    a. Definition: One of the most powerful ways to make a program readable is to break the calculations up into intermediate values that are held in variables with meaningful names, every name used in in the program should help with the readability of the code.

    b. Project Use: show how variables were used to make the code more readable.

16. Prefer Polymorphism to If/Else or Switch/Case

    a. Definition: There may be no more than one switch statement for a given type of selection. The cases in that switch statement must create polymorphic objects that take the place of other switch statements in the rest of the system.

    b. Project Use: I used if statements to differ different type of instances and the power of Polymorphism when creating the command pattern that we discussed earlier instead of switch statements. Refer to the figure below for the demonstration of the two cases.

17. Follow Standard Conventions

    a. Definition: Every team should follow a coding standard based on common industry norms. This coding standard should specify things like where to declare instance variables, how to name classes', methods, variables, and where to put braces and so on.

    b. Project Use: While coding this project I followed the Google Java Styling convention which we will discuss furthermore in the chapter that discusses the topic.

18. Replace Magic Numbers with Named Constants

a. Definition: In general, it is a bad idea to have raw numbers in your code. you should hide them behind well-named constants.

b. Project Use: The figure below will demonstrate the code before and after this rule has been used.

19. Be Precise

a. Definition: When you decide in your code, make sure your decision is precise. Know why you have made it and how you will deal with any exceptions.

b. Project Use: One of the use cases of this rule is calling a function that might return null we should for the null value, the figure below demonstrates an example.

21. Avoid Negative Conditionals

a. Definition: Negatives are just a bit harder to understand than positives. So, when possible, conditionals should be expressed as positives.

b. I will try to make it better.

22. Functions Should Do One Thing

a. Definition: It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than one thing, and should be converted into many smaller functions, each of which does one thing.

b. Project Use: Following the Single Responsibly Principle all functions in this project are single responsibility that should alter or execute for just a single piece of that program's functionality.

23. Hidden Temporal Couplings

a. Definition: Structure the arguments of your functions such that the order in which they should be called is obvious.

b. Project Use: yes I did.

24. Don't Be Arbitrary

a. Definition: Have a reason for the way you structure your code, and make sure that reason is communicated by the structure of the code. If a structure appears consistently throughout the system, others will use it and preserve the convention.

b. Project Use :I will try to do better.

25. Encapsulate Boundary Conditions

a. Definition: Boundary conditions are hard to keep track of. Put the processing of them in one place. Don’t let them leak all over the code

b. Project Use: Yes of course.

26. Keep Configurable Data at High Levels.

a. Definition: Constants and Configuration Parameters should be kept at the highest level possible. They should reside in a very high level and easy to change since they are passed into the lower levels of the application.

b. Project Use: I didn’t use it.

27. Avoid Transitive Navigation

a. Definition: In general, we don’t want a single module to know much about its collaborators. More specifically, if A collaborates with B, and B collaborates with C, we don’t want modules that use A to know about C.

b. Project Use: yes of course.

8.2.5 JAVA

1. Avoid Long Import Lists by Using Wildcards

a. Definition: Long lists of imports are daunting to the reader. We don’t want to clutter up the tops of our modules with 80 lines of imports. Rather we want the imports to be a concise statement about which packages we collaborate with.

b. Project Use: Since I used Google styling guide and this rule is the opposite of the rule mentioned in the google styling guide, I used the latter when importing the lists as it imports only the specific ones.

2. Don't Inherit Constants

a. Definition: Writing constants in an interface and then gaining access to those constants by inheriting that interface.

b. Project Use: No constants have been put into the class and left of to be inherited and accessed.

3. Constants versus Enums

a. Definition: You should use enums instead of constants

b. Project Use: You can refer to the figure below for an example on using enums instead of constants in the project.

```
public enum Role {
    ADMINISTRATOR,USER;
}
```

8.2.6 NAMES

1. Choose Descriptive Names

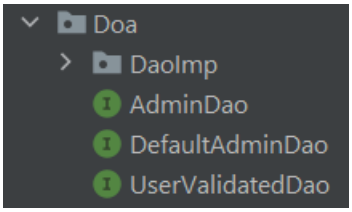
a. Definition: Names in software are 90 percent of what make software readable. You need to take the time to choose them wisely and keep them relevant. Names are too important to treat carelessly

b. Project Use: Referring to the project, the reader can see that the names in the project describe their use or their functions.

2. Use Standard Nomenclature Where Possible

a. Definition: Names are easier to understand if they are based on existing convention or usage. For example, if you are using the Decorator pattern, you should use the word Decorator in the names of the decorating class.

b. Project Use: Naming conventions are one of the most important in writing readable software, you can refer to the figure below for an example.



3. Unambiguous Names

a. Definition: Choose names that make the working of the function or variable unambiguous.

b. Project Use: Referring to the project, the reader can see that the names in the project describe their use or their functions.

4. Use Long Names for Long Scopes

a. Definition: The length of a name should be related to the length of the scope. You can use very short variable names for tiny scopes, but for big scopes you should use longer names.

b. Project Use: yes of course.

5. Avoid Encodings

a. Definition: Names should not be encoded with type of scope information. Prefixes such as m_ or f are useless in today's environments.

b. Project Use: No encodings have been used when I named the project classes functions and variables.

6. Names Should Describe Side-Effects

a. Definition: Names should describe everything that a function, variable, or class is or does. Don't hide the effects with a name. Don't use a simple verb to describe a function that does more than just a simple action.

b. Project Use: Refer to the figure below for an example in the project.

```
@Override
public boolean addDataBase(String dataBaseName) {

    boolean isCreated = dataBase.createDatabase(adminPath,dataBaseName);
    return isCreated;
}
```

8.2.7 TESTS

I didn't make the all test cases.

9 EFFECTIVE JAVA ITEMS (JUSHUA BLOCH)

9.1 Introduction

In this chapter we will be going through all the Code Principles that are mentioned in Joshua Bloch's Effective Java book, you as the reader could refer to any of the Principles discussed here and read more about it in the mentioned book.

I am sorry for not making a lot of principles

9.2 Code Principles.

9.2.1 CREATING AND DESTROYING OBJECTS.

1. Use Static Factory Methods instead of constructors.

    a. Definition: A Static factory method is simply a static method that returns an instance of the class.

2. Enforce the singleton property with a private constructor or an enum type.

    a. Definition: A singleton is simply a class that is instantiated exactly once [Gamma95]. Singletons typically represent either a stateless object such as a function (Item 24) or a system component that is intrinsically unique.

3. Prefer Dependency injection to Hardwiring.

    a. Definition: When a class might have many variants, you should pass the resource into the constructor when creating a new instance, this allows for an easier code scalability when new requirements are needed in the project.

4. Avoid Creating Unnecessary object.

    a. Definition: It is often appropriate to reuse a single object instead of creating a new functionally equivalent object each time it is needed. Reuse can be both faster and more stylish. An object can always be reused if it is immutable.

5. Avoid finalizers

a. Definition: Finalizers are unpredictable, often dangerous, and generally unnecessary. Their use can cause erratic behavior, poor performance, and portability problems.

b. Project Use: When designing this project, I never used finalizers and tried to avoid them at all cost.

9.2.2 METHODS COMMON TO ALL OBJECTS.

1. Obey the general contract when overriding equals.

a. Definition: You should avoid overriding equals in these situations. Each instance of the class is inherently unique like Threads, you don't care whether the class provides a "logical equality" test. like the java.util.Random A superclass has already overridden equals, and the superclass behavior is appropriate for this class or for the last point the class is private or package-private, and you are certain that its equals method will never be invoked

b. Project Use: I didn’t use it.

2. Always override hashCode when you override equals

a. Definition: You must override hashCode in every class that overrides equals. If you fail to do so, your class will violate the general contract for hashCode.

b. Project Use: i didn’t do it.

3. Always override toString

a. Definition: Always provide programmatic access to all of the information contained in the value returned by toString so the users of the object don't need to parse the output of the toString

b. Project Use: When designing this project, I tried to override the toString method in every place the developer might need to get information about the object.

9.2.3 CLASSES AND INTERFACES.

1. Minimize the accessibility of classes and members

a. Definition: A well-designed component hides all its implementation details, cleanly separating its API from its implementation.

b. Project Use: I tried to do that.

2. In Public Classes use accessor methods and not public fields.

    a. Definition: Degenerate classes should not be public and should be replaced by accessor methods (getters) and mutators (setters).

    b. Project Use: I tried to do that.

3. Minimize Mutability

    a. Definition: An immutable class is simply a class whose instances cannot be modified. All the information contained in each instance is fixed for the lifetime of the object, so no changes can ever be observed.

    b. Project Use: No immutable classes have been used in this project.

4. Favor composition over inheritance

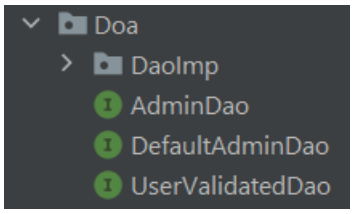
    a. Definition: Inheritance in this case is when a class extends another (implementation inheritance) Not interface inheritance and instead of extending, we give our new class a private field that references an instance of the existing class.

    b. Project Use: yes always.

5. Prefer Interfaces to Abstract Classes

    a. Definition: Because Java permits only single inheritance, this restriction on abstract classes severely constrains their use as type definitions. Any class that defines all the required methods and obeys the general contract is permitted to implement an interface, regardless of where the class resides in the class hierarchy.

    b. Project Use: Interfaces is generally the best way to define a type that permits multiple implementations where Existing classes can be easily retrofitted to implement a new interface and Interfaces allow the construction of nonhierarchical type frameworks, more over Interfaces enable safe, powerful functionality enhancements. For this principle you could refer to the following figure for an example.



9.2.4 GENERICS

1. Don't use raw types in new code

    a. Definition: If you use raw types, you lose all the safety and expressiveness benefits of generics.

    b. Project use: I didn’t need to use raw types in my code.

2. Eliminate unchecked warnings

    a. Definition: Eliminate every unchecked warning that you can, if you can´t use Suppress-Warnings annotation on the smallest scope possible.

    b. Project use: No warnings have been left off in the code.

3. Prefer lists to arrays

    a. Definition: One of the major differences between arrays and generics is that arrays are reified. This means that arrays know and enforce their element type at runtime. As noted earlier, if you try to put a String into an array of Long, you’ll get an “ArrayStoreException”. Generics, by contrast, are implemented by erasure]. This means that they enforce their type constraints only at compile time and discard (or erase) their element type information at runtime.

    b. Project use: yes I did it.

9.2.5 ENUMS AND ANNOTATIONS.

1. Use enumerators instead of integer constants

    a. Definition: An enumerated type is a type whose legal values consist of a fixed set of constants, such as the seasons of the year, the planets in the solar system, or the suits in a deck of playing cards. It provides type safety and expressive power more than the classic way of using integer enumerators pattern where we used to declare integer values to represent enumerators

    b. Project use: You can refer to the figure below for an example on enumerators where is used it to represent the Administrator .

```
String[] data = line.split( regex: ",");
if (data[0].equals(userName) && data[2].equals(String.valueOf(Role.ADMINISTRATOR))) {
    this.role= Role.valueOf(data[2]);
    return true;
}
```

9.2.6 METHODS.

1. Check parameters for validity

    a. Definition: Functions should have a small number of arguments. No argument is best, followed by one, two, and three.

    b. Project use: The only place the admin or user will be inputting data into the controller is when the controller takes the commands from the user, and this part is handled in a try catch block that I will give an example later in this documentation.

2. Design method signatures carefully

    a. Definition: Names should always obey the standard naming conventions for example the Oracle's Java Coding Style or Googles Java Style. Your primary goal should be to choose names that are understandable and consistent with other names in the same package.

    b. Project Use: Referring to the project you can see that all packages, classes, methods, and variables are all using the Google Style.

3. Return empty arrays or collections, not nulls

    a. Definition: There is no reason ever to return null from an array- or collection-valued method instead of returning an empty array or collection Return an immutable empty array instead of null.

    b. Project Use: Referring to the project there are no arrays or collection valued methods that return a null, some methods that are not collections that depend on null objects are returned for the purpose of checking their invalidity, we could use the Null object pattern, but it was not implemented in this project.

9.2.7 GENERAL PROGRAMMING.

1. Minimize the scope of local variables.

    a. Definition: This Principle is similar in nature to "Minimize the accessibility of classes and members." By minimizing the scope of local variables, you increase the readability and maintainability of your code and reduce the likelihood of error.

    b. Project Use: The most powerful technique for minimizing the scope of a local variable is to declare it where it is first used. This rule can be found in many places of the project.

2. Prefer for-each loops to traditional for loops.

    a. Definition: The Enhanced for loop officially known “for-each” gets rid of the clutter and the opportunity for error by hiding the iterator or index variable. The resulting idiom applies equally to collections and arrays, easing the process of switching the implementation type of a container from one to the other, but the downfall with foreach if you want to manipulate the indexes in a more advanced way you will lose that.

    b. Project Use: When coding the project, I only used loops to manipulate indexes and the for-each didn’t help me in it, so I didn’t use it.

3. Know and use libraries

    a. Definition: By using a standard library, you take advantage of the knowledge of the experts who wrote it and the experience of those who used it before you. So, you should always know the libraries and use them in your language of choice to ease and gain the mentioned advantage.

    b. Project Use: yes of course.

4. Avoid float and double if exact answer is required

    a. Definition: In summary, don’t use float or double for any calculations that require an exact answer.

    b. Project Use: I didn’t use it.

5. Prefer primitive types to boxed primitives

    a. Definition: Primitives are int, double, Boolean while Boxed Primitives are Integer, Double, Boolean. The problem is when we use “==” between boxed primitives it is almost always wrong.

    b. Project Use: I used only primitives in my project because I didn’t need Boxed ones.

6. Avoid Strings where other types are more appropriate

    a. Definition: Some reasons to avoid strings are

    i. more cumbersome than other types.

    ii. Strings are less flexible than other types.

iii. String are slower than other types.

b. Project Use: Referring to project you can see that I always used the appropriate type when needed.

7. Beware the performance of string concatenation

a. Definition: Using the string concatenation operator repeatedly to concatenate n strings requires time quadratic in n.

b. Project use: I will try.

8. Refer to objects by their interface

a. Definition: If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types. It makes the program much more flexible as we could change the implementation of the parameters, return values, variables and fields by changing just one line.

b. Project use: I will try.

9. Optimize judiciously

a. Definition: Strive to write good programs rather than fast ones, speed will follow. If a good program is not fast enough, its architecture will allow it to be optimized.

b. Project use: I will do my best.

10. Adhere to generally accepted naming conventions

a. Definition: You should always follow the coding conventions when coding, The Java platform has a well-established set of naming conventions, many of which are contained in The Java Language Specification.

b. Project use: Regrading this project I used Google's coding convention.

9.2.8 EXCEPTIONS.

1. Use exceptions only for exceptional conditions

a. Definition: Exceptions are, as their name implies, to be used only for exceptional conditions; they should never be used for ordinary control flow

b. Project use: No exceptions have been used in my code to control the flow of the program.

2. Favor the use of standard exceptions

    a. Definition: Using standard exceptions has several benefits.

    i. It makes your API easier to learn and use because it matches the established conventions that programmers are already familiar with.

    ii. Developers using your API are easier to read because they aren't cluttered with unfamiliar exceptions.

    iii. fewer exception classes means a smaller memory footprint and less time spent loading classes.

    b. Project use: In my project I only used standard exceptions in my try catch blocks.

3. Include failure-capture information in detail messages

    a. Definition: It is critically important that the exception's method return as much information as possible concerning the cause of the failure. This will help with resolving the issue.

    b. yes of course.

4. Don't ignore exceptions

    a. Definition: To keep it clear, never let catch blocks empty.

    b. Project use: No catch block is left empty in my code.

10 THE SOLID PRINCIPLES.

10.1 Introduction

Any developer that knows the syntax of his programming language could write code that satisfies present requirements. But writing code that could also satisfy future requirement easily should be the goal of every developer. Evolving with the time is the only factor which can keep you going, otherwise, you all know what happened with Nokia and Canon brands. SOLID principles help you to write such code. SOLID is a methodology for creating maintainable testable code. It's an acronym coined by the author Robert Martin (uncle Bob), we will be going through the five pillars of the SOLID methodology explaining them and providing an example from our project.

10.2 The Solid Principles.

10.2.1 S- SINGLE RESPONSIBILITY PRINCIPLE

1. Definition: Class should have only one responsibility that implies a class should be highly cohesive and implement strongly related logic. Class implementing feature 1 AND feature 2 AND feature 3 and so on violates the SRP Principle.

2. How to Recognize the code smell.:

    a. More than one contextually separated piece of code within single class.

    b. Large setup in tests (TDD is very useful when it comes to detecting SRP violation).

3. Benefits:

    a. Separated classes responsible for given use case can be now reused in other parts of an application.

    b. Separated classes responsible for given use case can be now tested separately.

4. Project Example: I tried to do it.

10.2.2 O- OPEN/CLOSED PRINCIPLE

1. Definition: Class should be open for extension and closed for modification. You should be able to extend class' behavior without the need to modify its implementation this can be achieved by not modifying existing code of class X but write a new piece of code that will be used by class X.

2. How to Recognize the code smell:

a. If you notice class X directly references other class Y from within its code base, it's a sign that class Y should be passed to class X (either through constructor/single method) e.g. through dependency injection

b. complex if-else or switch statements

3. Benefits:

a. Class' X functionality can be easily extended with new functionality encapsulated in a separate class without the need to change class' X implementation (it's not aware of introduced changes)

b. Code is loosely coupled c. Injected class Y can be easily mocked in tests

10.2.3 L- LISKOV SUBSTITUTION PRINCIPLE

1. Definition: Extension of open/closed principle stating that new derived classes extending the base class should not change the behavior of the base class (behavior of inherited methods. Provided that if a class Y is a subclass of class X any instance referencing class, X should be able to reference class Y as well. In other words, derived types must be completely substitutable for their base types.

2. How to Recognize the code smell:

a. If it looks like a duck, quacks like a duck but needs batteries for that purpose - it's probably a violation of LSP.

b. Modification of inherited behavior in subclass.

c. Exceptions raised in overridden inherited methods. 3. Benefits: a. Avoiding unexpected and incorrect results.

b. Clear distinction between shared inherited interface and extended functionality.

4. I tried to do it.

10.2.4 I- INTERFACE SEGREGATION PRINCIPLE

1. Definition: This principle suggests that "many client specific interfaces are better than one general interface". This is the first principle which is applied on interface, all the above three principles apply on classes. In other words, the client should not depend on interface which it is not going to be used.

2. How to Recognize the code smell: a. One fat interface implemented by many classes where none of these classes implement 100% of interface's methods. Such fat interface should be split into smaller interfaces suitable for client needs.

3. Benefits:

a. Increases the readability and maintainability of our code. We are reducing our class implementation only to required actions without any additional or unnecessary code.

b. Avoiding coupling between all classes using a single fat interface (once a method in the single fat Interface gets updated, all classes - no matter they use this method or not - are forced to update accordingly).

c. Clear separation of business logic by grouping responsibilities into separate interfaces

10.2.5 D- DEPENDENCY INVERSION PRINCIPLE

1. Definition: This principle suggests that "classes should depend on abstraction but not on concretion". It means that we should be having object of interface which helps us to communicate with the concrete classes.

2. How to Recognize the code smell:

    a. Instantiation of low-level modules in high-level ones.

    b. Calls to class methods of low-level modules/classes.

3. Benefits:

a. Increase reusability of higher-level modules by making them independent of lower-level modules

b. Injected class can be easily mocked in tests.

c. Loosely couple architecture.

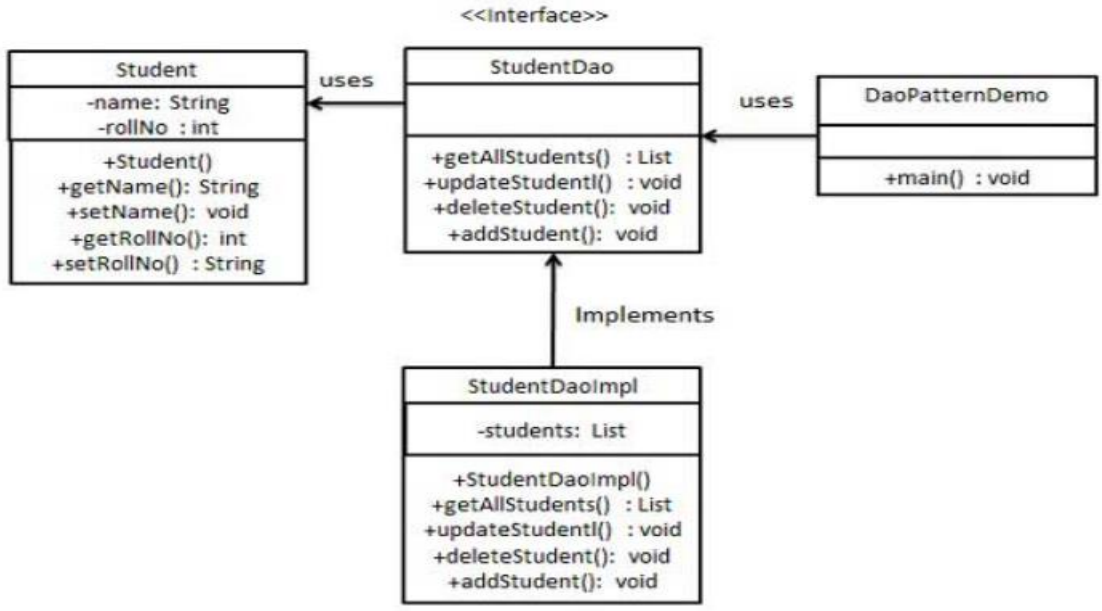
4. Project Example: I didn't use it.

11 DESIGN PATTERNS.

DAO:

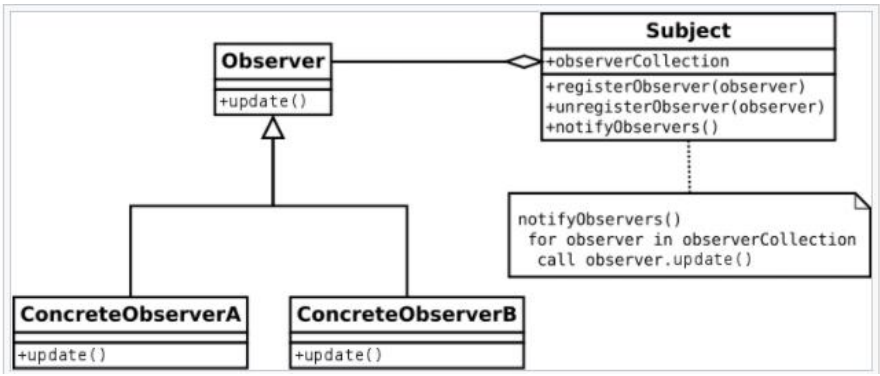
DAO stands for Data Access Object. DAO Design Pattern is used to separate the data persistence logic in a separate layer. This way, the service remains completely in dark about how the low-level operations to access the database is done. This is known as the principle of Separation of Logic

Example:



Observer

The Observer design pattern is a behavioral pattern, among the twenty-three well-known "Gang of Four" design patterns describing how to solve recurring design challenges in order to design flexible and reusable object-oriented software, i.e. objects which are easier to implement, change, test, and reuse.



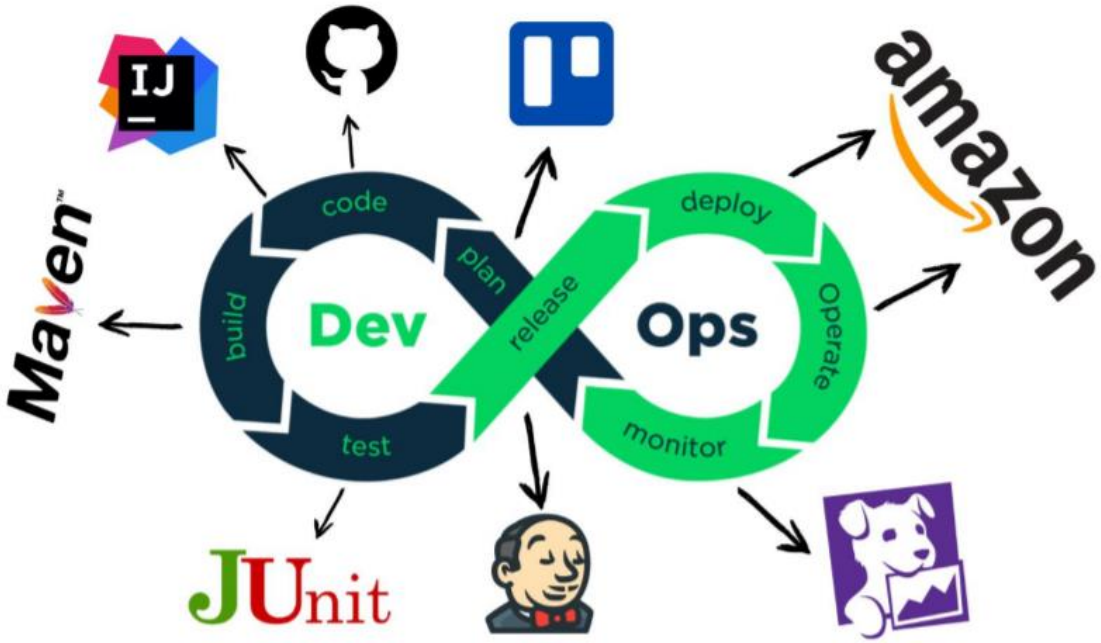
12 DEVOPS PRACTICES.

12.1 Introduction

In this chapter ill talk about how I implemented the system, the protocol between the client and server, DevOps, how design patterns are used. Data structures implemented, how did I tried as much as possible to abide by the rules of clean code, SOLID principles, effective java and ACID criteria.

12.2 DevOps

DevOps is all about implementing the best practices with the help of tools To Provide Continues delivery with high quality software, learning about DevOps was the most satisfactory learning process I have ever gone through. Jez Humble (co-author of the DevOps Handbook) describes it as “DevOps is not a Goal, But a never-ending process of continual improvement”. The more I learned about DevOps and the tools used, the more I felt hungry for more and more learning.



1. PLANNING

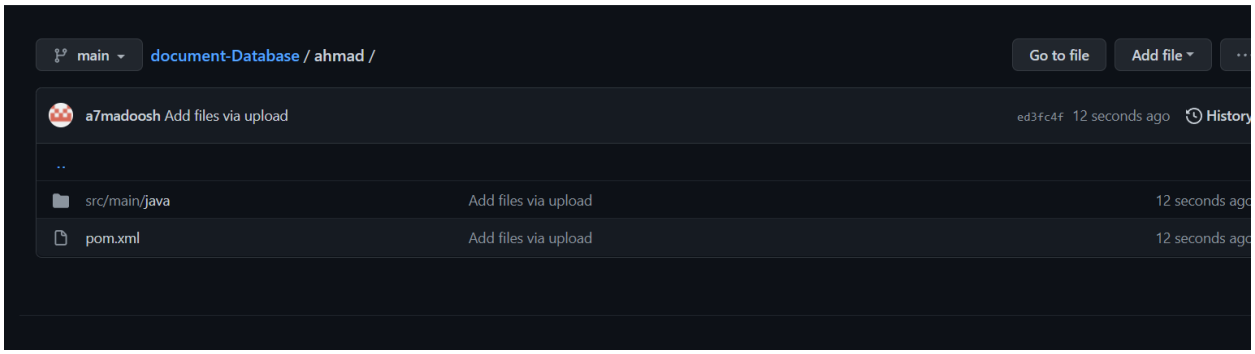
To be honest the plan didn't work and I took a lot of time to understand what we want to build.

2. INTELLIJ ULTIMATE

IntelliJ Ultimate is one of the best IDEs for java development, its an excellent software. I have used eclipse and its not even close to IntelliJ in Terms of performance.

3. GITHUB

For me whenever someone mentions Version control the first thing that comes to my mind is GitHub, its known to everyone that GitHub is the best version control system, it's the best platform to store our code without any disturbance.



4. MAVEN

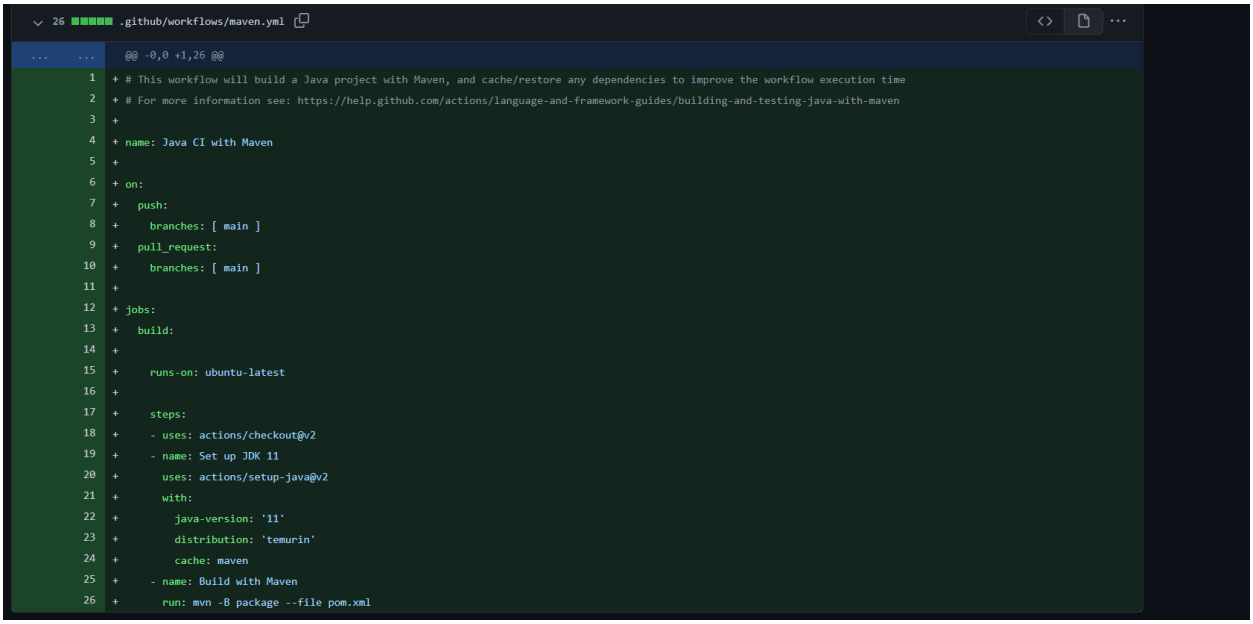
Maven is a powerful build automation tool for Java projects, it can be used for any java project. It provides an easy way to build the project in which the complex details. are hidden with the help of dynamic downloading of plug-ins and libraries from the maven repository website.

5. JUNIT

In the testing phase of DevOps cycle, I used Junit framework, once a build succeeds and is deployed to the testing phase a series of unit testing are preformed, also Manual testing is performed.

6. GitHub ACTIONS

As GitHub Actions provides CI/CD, I think this one of the most important phases in DevOps pipeline, when we reach this stage that means that a build is ready and safe for deployment for the next stage. Since each code has passed a bunch of manual and unit testing. So, we will be confident about the deployment because the likelihood of issues or bugs are very low.



7. DOCKER

Docker is a software platform that enables software developers to easily integrate the use of containers into the software development process. The Docker platform is open source and available for Windows and Mac, making it accessible for developers working on a variety of platforms. The application provides a control interface between the host operating system and containerized applications.