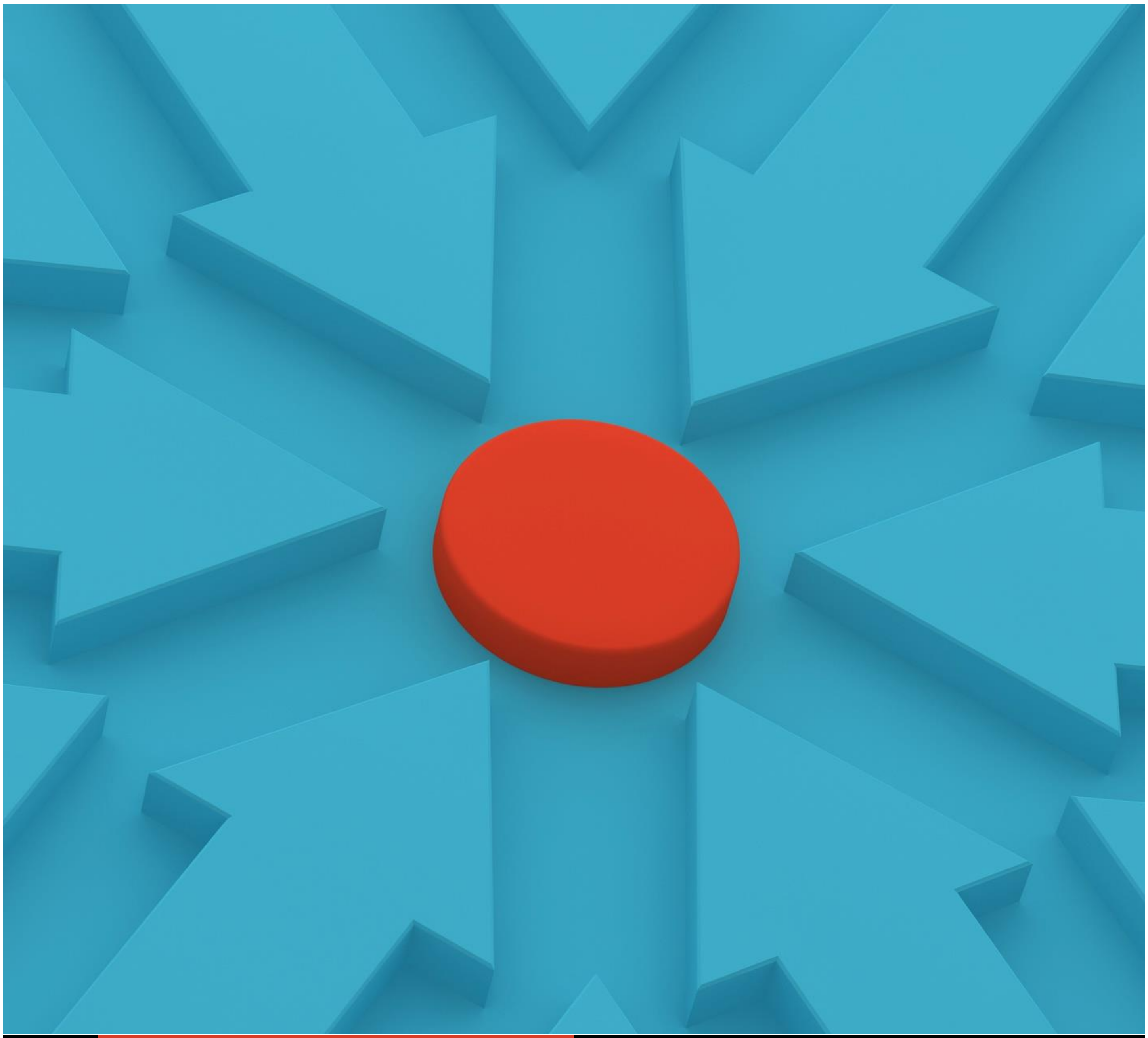


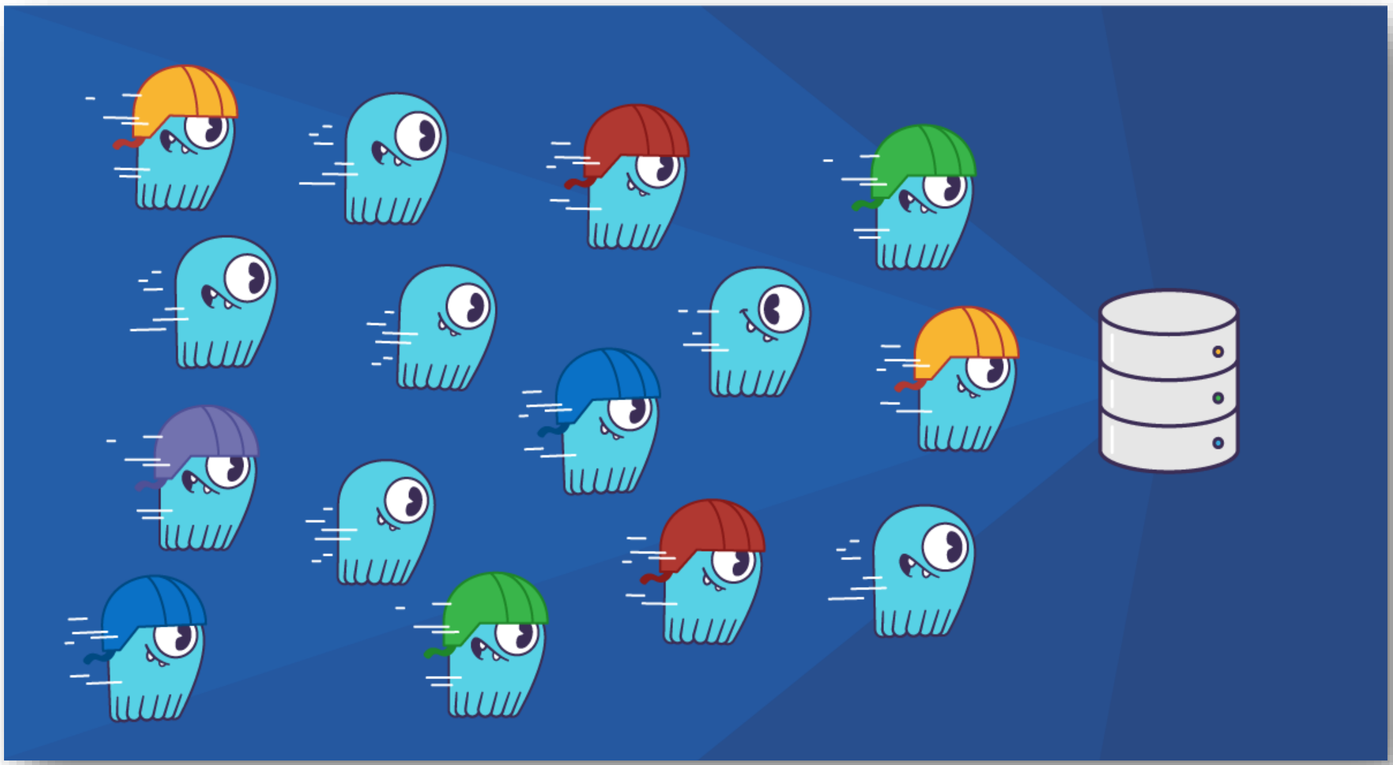
MULTI-THREADED DATABASE SERVER SYSTEM

OMAR RAHWANJI

Task #4

Table of Contents

Concurrency



Concurrency refers to the ability to run multiple tasks at the same time, which can increase the efficiency of an application. Tasks can be submitted to run immediately, at a specified time, or in response to the completion of one or more actions upon which they depend. For example, an application that computes the total cost of a merchandise order must run a series of tasks. First, it looks up the state and local sales tax rates in one task while concurrently calculating the shipping costs in another. Then, in a third task that runs upon completion of the first two, it totals the results. Java SE and EE both support concurrency with a standardized API that enables the creation and scheduling of concurrent tasks.

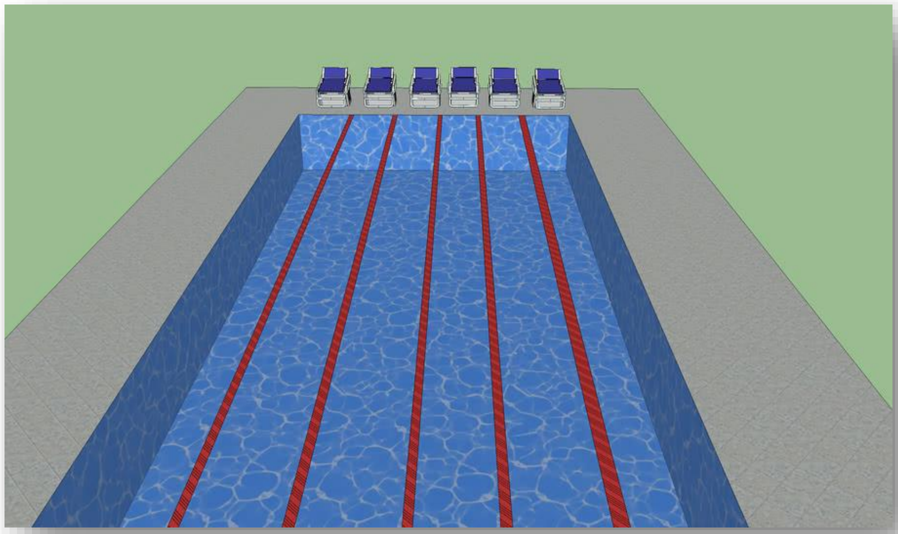
Multi-Threading

Multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Threads' Pool



Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks. An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread. While this approach seems simple to implement, it has significant disadvantages. A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.

Since active threads consume system resources, a JVM creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.

Thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing. Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

Java provides the Executor framework which is centered around the Executor interface, its sub-interface –ExecutorService- and the class -ThreadPoolExecutor-, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.

They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanic

Race Conditions

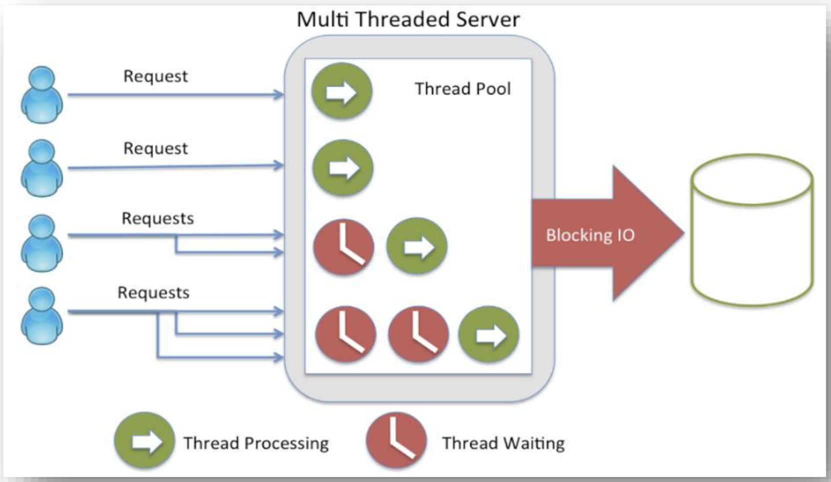
Race condition is a concurrency problem that may occur inside a critical section. A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition. The term race condition stems from the metaphor that the threads are racing through the critical section, and that the result of that race impacts the result of executing the critical section.

Synchronizing the execution of the critical section is the solution here. Locks can be used for preventing more than a single thread of executing the critical section.

Multi-Threaded Database Server

Server having more than one thread is known as Multithreaded Server. When a client sends the request, a thread is generated through which a user can communicate with the server. We need to generate multiple threads to accept multiple requests from multiple clients at the same time.



Advantages of Multithreaded Server:

- Quick and Efficient: Multithreaded server could respond efficiently and quickly to the increasing client queries quickly.

- Waiting time for users decreases: In a single-threaded server, other users had to wait until the running process gets completed but in multithreaded servers, all users can get a response at a single time so no user has to wait for other processes to finish.

- Threads are independent of each other: There is no relation between any two threads. When a client is connected a new thread is generated every time.

- The issue in one thread does not affect other threads: If any error occurs in any of the threads then no other thread is disturbed, all other processes keep running normally. In a single-threaded server, every other client had to wait if any problem occurs in the thread.

Disadvantages of Multithreaded Server:

- Complicated Code: It is difficult to write the code of the multithreaded server. These programs cannot be created easily.

- Debugging is difficult: Analyzing the main reason and origin of the error is difficult.

Caching the Database Server

Caching is the process of storing the results of a request at a different location than the original or a temporary storage location so that we can avoid redoing the same operations. Basically, the cache is temporary storage for files and data such that it's faster to access this data from this new location.

Eviction policy:

We need to delete existing items for new resources when the cache is complete. In fact, it is just one of the most popular methods to delete the least recently used object. The solution is to optimize the probability in the cache that the requesting resource exists.

- Random Replacement (RR):

As the term suggests, we can randomly delete an entry.

- Least frequently used (LFU):

We can keep a count of how frequently an item is requested and delete the least frequently used.

- Least Recently Used (LRU):

In LRU, we delete the item that has been used least recently.

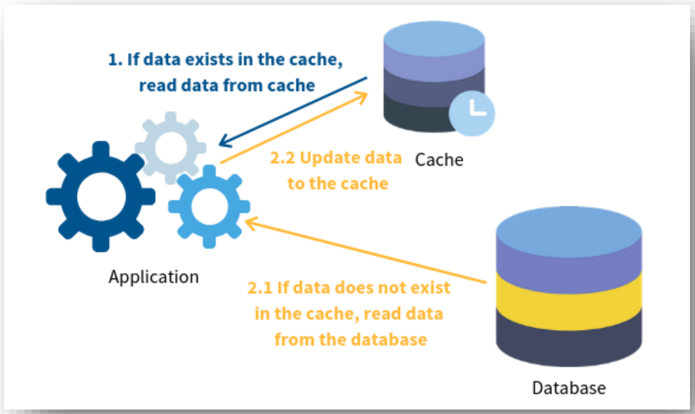
- First in First out (FIFO):

The FIFO algorithm holds an object queue in the order that the objects have been loaded into the cache. It evicts one or more objects from the head when a cache misses and inserts a new object into the queue tail. Upon a cache hit, the list does not shift.

Caching Strategies

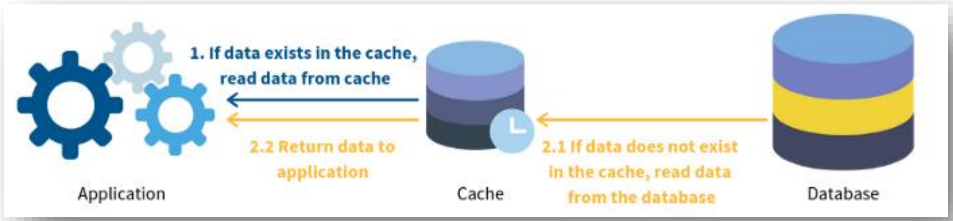
- Cache Aside:

In this strategy, the cache is sitting aside the database. The application will first request the data from the cache. If the data exists (we call this a 'cache hit'), the app will retrieve the data directly. If not (we call this a 'cache miss'), the app will request data from the database and write it to the cache so that the data can be retrieved from the cache again next time.


1. If data exists in the cache, read data from cache
2.2 Update data to the cache
Cache
Application
2.1 If data does not exist in the cache, read data from the database
Database

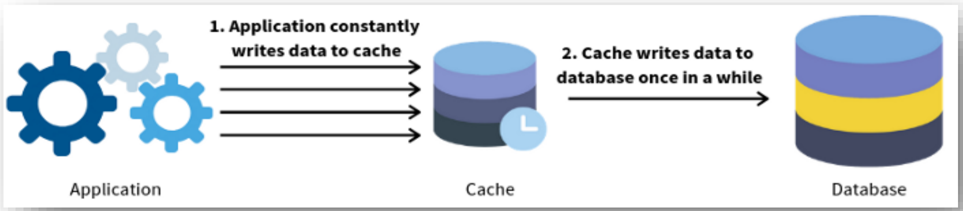
- Read Through:

Unlike cache aside, the cache sits in between the application and the database. The application only request data from the cache. If a 'cache miss' occurs, the cache is responsible to retrieve data from the database, update itself and return data to the application.

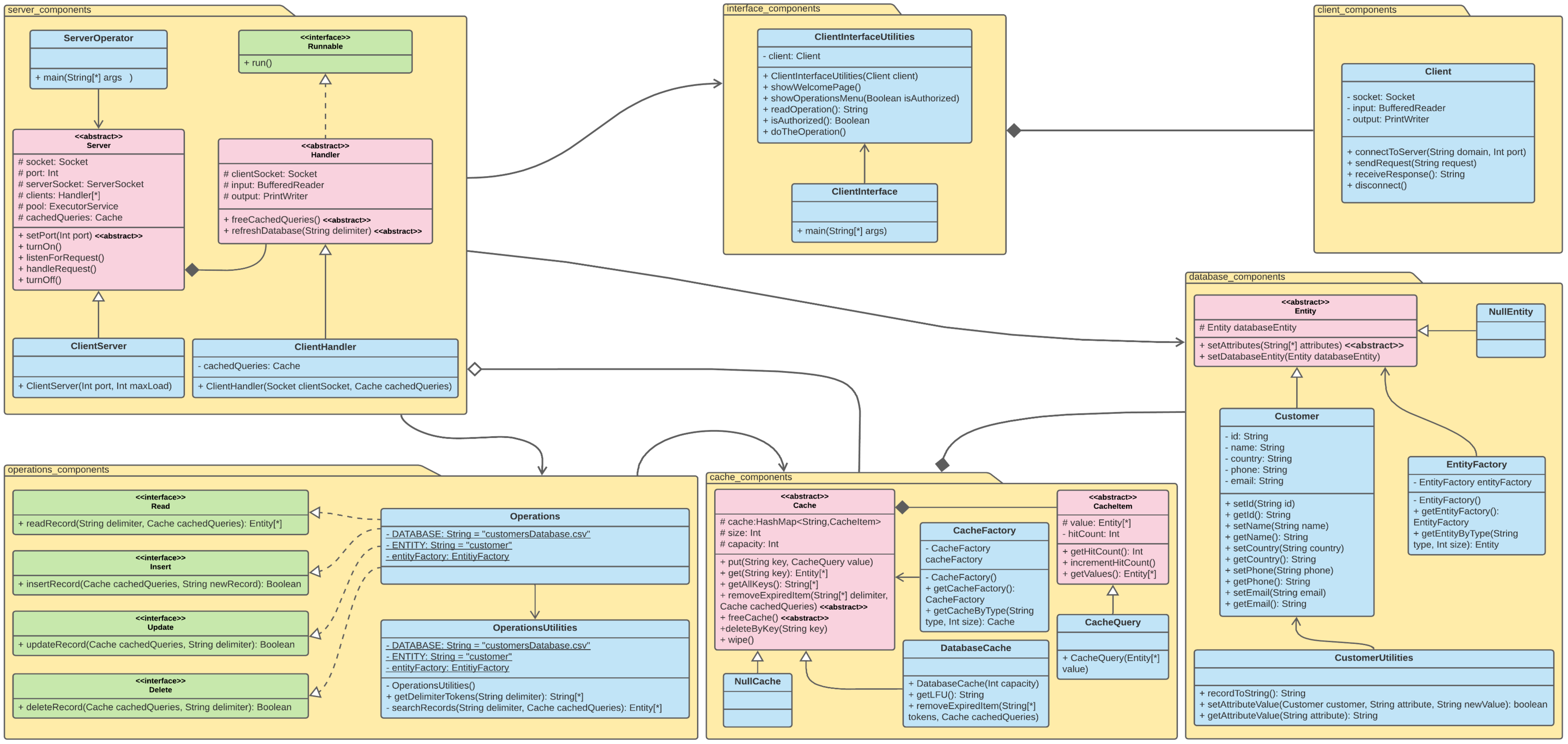

1. If data exists in the cache, read data from cache
2.2 Return data to application
2.1 If data does not exist in the cache, read data from the database
Application
Cache
Database

- Write Back (also known as Write Behind):

The application still writes data to the cache. However, there is a delay in writing from the cache to the database. The cache only flushes all updated data to the DB once in a while (e.g. every 2 minutes).


1. Application constantly writes data to cache
2. Cache writes data to database once in a while
Application
Cache
Database

Package-level Class Diagram of the System


server_components
ServerOperator
+ main(String[*] args)
<<interface>> Runnable
+ run()
<<abstract>> Server
# socket: Socket
# port: Int
# serverSocket: ServerSocket
# clients: Handler[*]
# pool: ExecutorService
# cachedQueries: Cache
+ setPort(Int port) <<abstract>>
+ turnOn()
+ listenForRequest()
+ handleRequest()
+ turnOff()
<<abstract>> Handler
# clientSocket: Socket
# input: BufferedReader
# output: PrintWriter
+ freeCachedQueries() <<abstract>>
+ refreshDatabase(String delimiter) <<abstract>>
ClientServer
+ ClientServer(Int port, Int maxLoad)
ClientHandler
- cachedQueries: Cache
+ ClientHandler(Socket clientSocket, Cache cachedQueries)
interface_components
ClientInterfaceUtilities
- client: Client
+ ClientInterfaceUtilities(Client client)
+ showWelcomePage()
+ showOperationsMenu(Boolean isAuthorized)
+ readOperation(): String
+ isAuthorized(): Boolean
+ doTheOperation()
ClientInterface
+ main(String[*] args)
client_components
Client
- socket: Socket
- input: BufferedReader
- output: PrintWriter
+ connectToServer(String domain, Int port)
+ sendRequest(String request)
+ receiveResponse(): String
+ disconnect()
database_components
<<abstract>> Entity
# Entity databaseEntity
+ setAttributes(String[*] attributes) <<abstract>>
+ setDatabaseEntity(Entity databaseEntity)
NullEntity
Customer
- id: String
- name: String
- country: String
- phone: String
- email: String
+ setId(String id)
+ getId(): String
+ setName(String name)
+ getName(): String
+ setCountry(String country)
+ getCountry(): String
+ setPhone(String phone)
+ getPhone(): String
+ setEmail(String email)
+ getEmail(): String
EntityFactory
- EntityFactory entityFactory
- EntityFactory()
+ getEntityFactory(): EntityFactory
+ getEntityByType(String type, Int size): Entity
CustomerUtilities
+ recordToString(): String
+ setAttributeValue(Customer customer, String attribute, String newValue): boolean
+ getAttributeValue(String attribute): String
operations_components
<<interface>> Read
+ readRecord(String delimiter, Cache cachedQueries): Entity[*]
<<interface>> Insert
+ insertRecord(Cache cachedQueries, String newRecord): Boolean
<<interface>> Update
+ updateRecord(Cache cachedQueries, String delimiter): Boolean
<<interface>> Delete
+ deleteRecord(Cache cachedQueries, String delimiter): Boolean
Operations
- DATABASE: String = "customersDatabase.csv"
- ENTITY: String = "customer"
- entityFactory: EntityFactory
OperationsUtilities
- DATABASE: String = "customersDatabase.csv"
- ENTITY: String = "customer"
- entityFactory: EntitiyFactory
- OperationsUtilities()
+ getDelimiterTokens(String delimiter): String[*]
- searchRecords(String delimiter, Cache cachedQueries): Entity[*]
cache_components
<<abstract>> Cache
# cache:HashMap<String,CacheItem>
# size: Int
# capacity: Int
+ put(String key, CacheQuery value)
+ get(String key): Entity[*]
+ getAllKeys(): String[*]
+ removeExpiredItem(String[*] delimiter, Cache cachedQueries) <<abstract>>
+ freeCache() <<abstract>>
+deleteByKey(String key)
+ wipe()
CacheFactory
- CacheFactory cacheFactory
- CacheFactory()
+ getCacheFactory(): CacheFactory
+ getCacheByType(String type, Int size): Cache
<<abstract>> CacheItem
# value: Entity[*]
- hitCount: Int
+ getHitCount(): Int
+ incrementHitCount()
+ getValues(): Entity[*]
CacheQuery
+ CacheQuery(Entity[*] value)
NullCache
DatabaseCache
+ DatabaseCache(Int capacity)
+ getLFU(): String
+ removeExpiredItem(String[*] tokens, Cache cachedQueries)

System Description

The diagram above depicts our Multi-Threaded In-Memory Database System. As we can see above, the system is composed of the following packages:

1) Server components, and it consists of the following classes:
   a- Server, which is an abstract class, and it is responsible for configuring the server.
   b- Client server, which is a subtype of a Server.
   c- Handler, which is an abstract class & it implements the runnable interface to give it the ability of being passed into threads.
   d- Client handler, which is a subtype of a Handler, and it is responsible for handling all of the clients' requests.
   e- Server operator, which is responsible for turning on the server.

2) Client components, and it consists of a single class, which is responsible for establishing the connection with the server. Also, it is responsible for sending and accepting requests from the server.

3) Database components, and it consists of the following classes:
   a- Entity, which is an abstract class, and it represents our in real world object to store its attribute in our database.
   b- Customer, which is a subtype an entity, and it has the following attributes: id, name, country, phone, email. Its primary key is the id.
   c- Customer utilities, which is providing some helpful tools to manipulate the attributes of the customer.

4) Operation components, and it consists of the following classes:
   a- Read, which is an interface, and it is responsible for retrieving multiple records from the cache (cache-hit), or the database (cache-miss) based on the condition, which is provided by the user.
   b- Insert, which is an interface, and it is responsible for adding a single record at a time to the database.
   c- Update, which is an interface, and it is responsible for editing multiple records by searching for them in the cache (cache-hit), or in the database (cache-miss), then update their attributes and finally, replacing them in both of the database and the cache.
   d- Delete, which is an interface, and it is responsible for deleting multiple records by searching for them in the cache (cache-hit), or in the database (cache-miss), then deleting them from both of the database and the cache.
   e- Operations, which is responsible for implementing the CRUD operations discussed above. Moreover, it is extensible so whenever a new operation needed to be added it will be implemented with ease!

5) Cache components, and it contains the following classes:
   a- Cache, which is responsible for holding our records in the memory using a hash table, for enhancing the speed of the CRUD operations by a huge margin. The hash table is storing the CRUD operation as a key, and its result as a list of values (records).
   b- Cache item, which is an abstract class, and it represents our value to be cached in the cache. Moreover, it is holding the number of cache hits of it so we can implement the Least frequently used (LFU) eviction policy for the cache.
   c- Cache query, which is representing our query/result to be cached.

6) Interface components, and it contains the following classes:
   a- Client interface, which is responsible for managing the interface for the user.
   b- Client interface utilities, which is responsible for offering some helpful tools to support the process of showing the user interface to the user.

A tour in our Caching Strategy

Let us begin with the data-structures of the cache itself. Our cache is basically a hash map, where its key is a string & its value is a custom data-structure I've made. I've decided to cache queries' results instead of caching individual records. So, the key string of the cache is going to hold the delimiter of the returned records based on the CRUD operation, While the value of each query is a CacheQuery object, which is a subtype of CacheItem class and it holds a list of the returned records, as well as a hitting count, which serves well for our hit-based eviction policy.

Our caching policy, which is known as “Cache Aside”, is all about sitting the cache aside the database. The application will first request the data from the cache. If the data exists (we call this a ‘cache hit’), the app will retrieve the data directly. If not (we call this a ‘cache miss’), the app will request data from the database and write it to the cache so that the data can be retrieved from the cache again next time.

Our main goal is to reduce the complexity & the time consumption of the read operation, which has the most demand in compare with other operations. Our cache will be storing each of the CRUD operations' results on the memory using a hash list so we can retrieve repeated operations with a blazing fast speed!

For the eviction policy, we will be using the least frequently used (LFU) policy for making some space to storing operations on the cache whenever it becomes full. Clearly, it is one of the best cache-eviction policies out there. It relays on the hit count of our stored queries to determine which of the cached queries should be evacuated. It helps keeping our cache active most of the time!

For the sake of controlling & preventing the race conditions, which occurs whenever multiple threads are accessing the same resource, we decided to put a lock on the process of refreshing the database, which would transfer our new modifications from the cache to the database. We saw that this is the most critical process in the system so by preventing the occurrence of it, our system will be thread-safe by far.

SOLID Principles in Action

One of the most if not the most important concept in software development is the SOLID principles. The word SOLID is constructed by taking the first letter in each of the six design principles, which are:

1) Single Responsibility Principle (SRP): a class should have only one reason to change (responsible for a one actor).

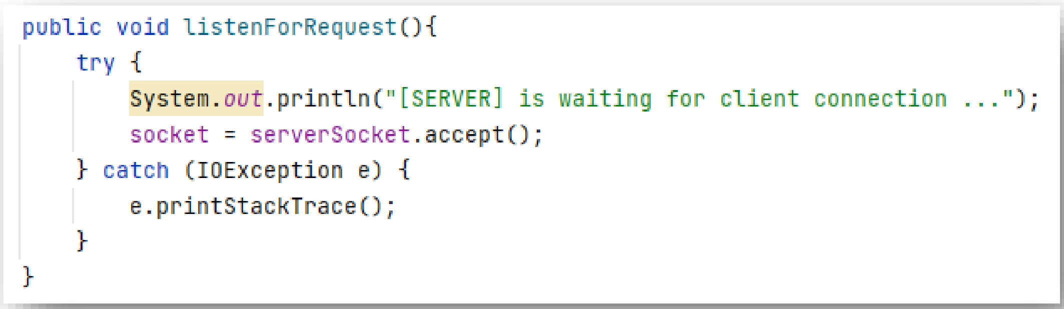
2) Open Close Principle (OCP): classes should be open for extension, but closed for modifications.

3) Liskov Substitution Principle (LSP): if z is subtype of Z, then all objects of type Z could be replaced with objects of type z (subtypes should be able to do what base types do).

4) Interface Segregation Principle (ISP): subtypes shouldn't be forced to inherit useless methods.

5) Dependency Inversion Principle (DIP): big modules shouldn't depend on small modules; they both should depend on abstract instead of concrete.

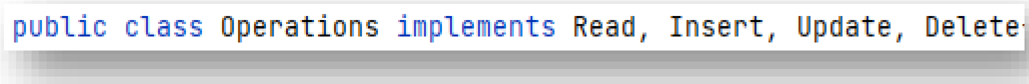
As for the SRP, each method in your class & each class in your module should be doing one & only one thing!



```java
public void listenForRequest(){
    try {
        System.out.println("[SERVER] is waiting for client connection ...");
        socket = serverSocket.accept();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Simplicity would be at your fingertips whenever you stick to this principle, just look at this snippet of code from our system, can't you imagine how simple it is? I won't say a word about its responsibility. Actually, you are capable of determining the functionality of it as you were the original writer of the code snippet!

As for the OCP, your modules should be extendable, not modifiable. Let me explain it to you with a code snippet;

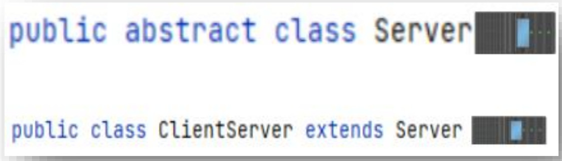


```
public class Operations implements Read, Insert, Update, Delete
```

Just take look on the code snippet. Simply, it is saying that our operations class will have its ability to perform the CRUD operations on its own, by its own style, version, whatever … you've got it. Imagine that there is a new pack of operations in the market and your boss asked you to support them in your system. If your module is built to be modifiable, then you'll end up missing with hundreds, if not thousands of lines of code, and guess what, you will break your system without knowing that! On the other hand, if your module were designed in an extendable way, then you can add any new feature to the system with ease, just implement your own final version of it. Do not modify it! Just write the best, the simplest version of it.

As for the LSP, it makes more sense that children are capable of doing what their parents do, so if your father is a programmer and can write simple code, it is more likely that you'll end up having the skill which he had! Enough chattering and let's bring a code snippet.

Since the small server is a subtype of the bigger one, so it should be capable of doing what the bigger server does. It should be able to send, receive, handle requests since it's called a server!

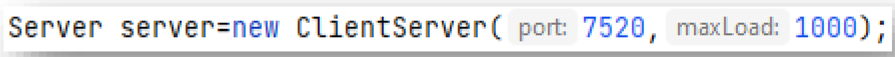


```
public abstract class Server

public class ClientServer extends Server
```

As for the ISP, what is the point of buying yourself useless heavy tools, which you'll be abandoning right after buying them. Your classes shouldn't be forced to inherit useless methods too! So, try to separate the parent class & the child class with an abstraction layer if the child class might not use some of the parent's methods.

As for the DIP, it is non-sense to design your house based on the shape of the windows, which you've bought from IKEA on sales! You should pick up some windows which can fit rightly in your house, not the other around. How much windows can cost in compare with a full-fledged house, make sense now?
By the way, you can achieve this principle using polymorphism.



```
Server server=new ClientServer( port: 7520, maxLoad: 1000);
```

As you can see in the code snippet, my server operator is depending on a server, it doesn't care about the type of the server. Instead, all what it cares about is just to being a server!

Design Patterns. A guide for the best practices

Design patterns help you avoid the head-ache of describing your system's functionalities to your crew-mates. With just a simple word, your list of ideas, which contains tons of code, could be understood. What a relief!

The reason behind that previous advantage is that design patterns are the best practices for well-known problems, which experts have faced before you and came out with the best, efficient, readable solution. So, please do not re-invent the wheel and just start using design patterns. Obviously, you cannot beat experts' designs by your designs, no matter you are pro. Also, it is good to follow an undiscussable reference like a particular design pattern.

Let me list some of the design patterns, which I've used in the system:

1- Singleton DP: [restricts the creation of objects]

Singleton pattern is a software design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system.

2- Factory Method DP: [eliminates the use of constructors (new)]

The factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object.

3- Null Object DP: [preventing null pointer exception]

A null object is an object with no referenced value or with defined neutral behavior. The null object design pattern describes the uses of such objects and their behavior.

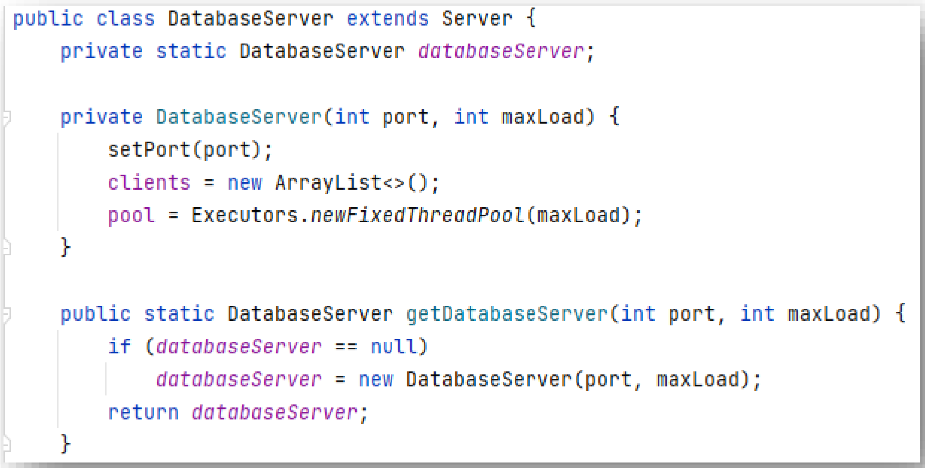
Design Patterns. In Action

As shown above, I've used some of the design patterns to help improving the system's code and forcing it to follow some of the experts' best practices.

1- Singleton DP:

I've used this design pattern in the DatabaseServer, CacheFactory and EntityFactory classes. Why, let me tell you by the following scenario;

Unfortunately, our company is poor. Poorly enough to barely getting a single database server. Allowing the IT department to get their hands on a second server would make our company fall apart financially!



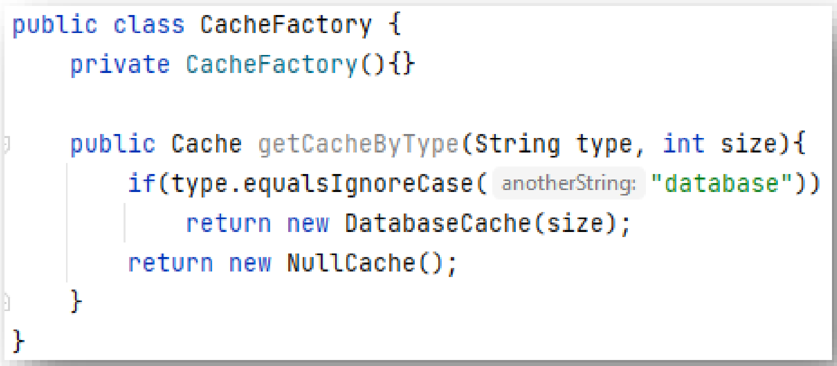
```
public class DatabaseServer extends Server {
    private static DatabaseServer databaseServer;

    private DatabaseServer(int port, int maxLoad) {
        setPort(port);
        clients = new ArrayList<>();
        pool = Executors.newFixedThreadPool(maxLoad);
    }

    public static DatabaseServer getDatabaseServer(int port, int maxLoad) {
        if (databaseServer == null)
            databaseServer = new DatabaseServer(port, maxLoad);
        return databaseServer;
    }
```

2- Factory Method DP:

I've used this design pattern to manage the creation of the entities & caches whenever they're needed. By using this design pattern, you can specify the type of the entity or the type of the cache needed with ease!

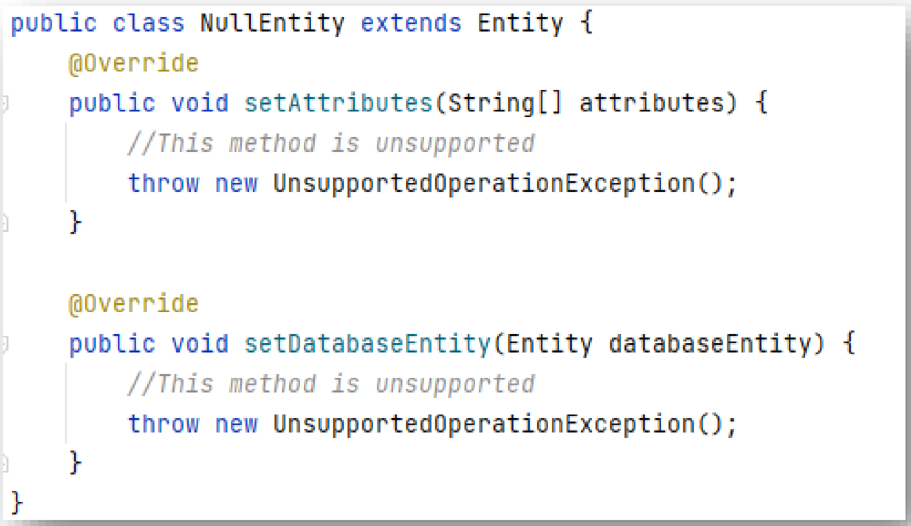


```
public class CacheFactory {
    private CacheFactory(){}

    public Cache getCacheByType(String type, int size){
        if(type.equalsIgnoreCase( anotherString: "database"))
            return new DatabaseCache(size);
        return new NullCache();
    }
}
```

3- Null Object DP:

I've used this design pattern to prevent odd actions from happening when entities or caches are used. Just imagine that you've provide the cache factory with a missing type be created. Instead of throwing exceptions or doing some odd actions, it will just tell you that this object is null, it doesn't even exist!



```java
public class NullEntity extends Entity {
    @Override
    public void setAttributes(String[] attributes) {
        //This method is unsupported
        throw new UnsupportedOperationException();
    }

    @Override
    public void setDatabaseEntity(Entity databaseEntity) {
        //This method is unsupported
        throw new UnsupportedOperationException();
    }
}
```

Finally, What about ACID?

ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other mishaps. In the context of databases, a sequence of database operations that satisfies the ACID properties (which can be perceived as a single logical operation on the data) is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

1) Atomicity:

Transactions are often composed of multiple statements. Atomicity guarantees that each transaction is treated as a single "unit", which either succeeds completely, or fails completely: if any of the statements constituting a transaction fails to complete, the entire transaction fails and the database is left unchanged. As for achieving this property, our system is designed in a way that the CRUD operations would affect the cache first, if no systematic failures occur, then the database should be refreshed with ease!

2) Consistency:

This property ensures that a transaction can only bring the database from one valid state to another, maintaining database invariants: any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This prevents database corruption by an illegal transaction, but does not guarantee that a transaction is correct. Referential integrity guarantees the primary key – foreign key relationship. As for achieving this property, our system's database contains a primary key, which is the id of the entity. Inserting a record with an existing id is a violation of the uniqueness rule of the primary key. Also, our system will prevent your from keeping the primary key value null, as this violates the not null constraint of the primary key.

3) Isolation:

Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time). Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially. Isolation is the main goal of concurrency control; depending on the method used, the effects of an incomplete transaction might not even be visible to other transactions. As for achieving this property, our system assume that the critical section is the block, which refresh the database. Thus, it locks it and prevent concurrent threads from accessing it.

4) Durability:

This property guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory. As for achieving this property, our system is capable of backing up some of the last CRUD operations done on the database, so it will be durable by far.