Exercises 2 & 3

In this exercise, there will be two main tasks:

- Implement a multi-threaded version of a chat system.
- Refactor the code from the previous exercise to improve its structure in order to accommodate changes to ChatSys in the next exercise sessions.

In addition to learning about refactoring and threads, the goal of these tasks is also to make you practice using interfaces and abstract classes which are two important object-oriented concepts in Java in general, particularly for thread and network programming.

Several class diagrams, one for each version of ChatSys described below can be found on Moodle: https://moodle.njit.edu.cn/moodle/mod/folder/view.php?id=2218

Note: you are **not required** to implement all these changes by the end of the exercise session 2 (Friday 23rd). Try to implement as much as you can, and during exercise 3 (Tuesday 27th) you will keep on working starting from where you finished. Also, you shouldn't work on ChatSys 0.4.4 and ChatSys 0.5 before the third lecture (Monday 26th) as these two exercises cover topics that haven't been introduced yet and will be in the next lecture.

ChatSys 0.4.0

For these two exercise sessions, the main goal is to implement a system where different threads can communicate with each other through a "server thread". In this version of ChatSys, there won't be any network communications. Instead, we will have different threads communicating with a server through message queues.

The clients will send messages to the server by adding messages in a message queue. When the server receives a message in the queue, it forwards the message to all the clients. Clients also have a message queue. It is used by the server to send (i.e. forward) messages to the clients. Clients **cannot communicate directly with each other** and have to communicate through the intermediate of the server.

MessageQueue

Because the server and all the clients will have their own message queue, we will start by creating a class MessageQueue in a new package programming3.chatsys.threads.

A message queue has an attribute "queue" of type Queue<ChatMessage>. Queue is an interface defined by the Java standard API. (LinkedList is an implementation of such Queue.)

The message queue will have two methods:

- send(ChatMessage message) that adds a chat message to the queue
- ChatMessage getMessage(int waitTime) that returns a message from the queue

The specificity of getMessage is that if there is no message in the queue, it will wait a certain amount of time and try reading from the queue again. For reading a message, you can use either the queue's poll or remove methods. If a message is successfully read, you should return it, otherwise you should wait a certain amount that is provided in milliseconds in the parameter. (You can wait using the method Thread.sleep.)

ThreadClient and ThreadServer

Now let's create two classes that inherits from MessageQueue: ThreadClient and ThreadServer.

These two classes need to implement the class "Runnable" so they can be run in separate threads.

Write a run method in each class that has an infinite loop and do the following in each iteration of the loop

- Read a message from the queue using the method getMessage method defined in MessageQueue.
- Catch the potential InterruptedException normally thrown by calling getMessage and break the loop so the thread can end properly if it is interrupted

When a message is successfully retrieved the server does the following:

- Add the message to the database
- Forward the message to the other clients (create a method "forwardMessage" that you will implement later)

When a message is successfully retrieved by the client, simply print it to the console with System.out.println.

For the server and the clients to be able to communicate with each other, we need to have a way for them to be aware of each other. For that, a client should have a class attribute storing a reference to the server that will

be provided in the constructor. We will also give to clients a class attribute name that is a String and has an associated getName method so we can more easily identify clients.

Then the server needs to have a class attribute for the Database interface that will be provided in its constructor, and an empty set of clients (i.e. of type Set<ThreadClient>). It also needs to have a method "register" that allows a client to register itself with the server (i.e. be added to the set of clients) and a method unregister for doing the opposite (i.e. remove itself from the set of clients).

Now we can also implement the method forward that will loop over all the clients part of the sets, and send them the message to forward.

Finally, when we start our client, before entering the loop waiting for messages, our client needs to register itself with the server. We will also make our client send a message "Hello World!" after registering with the server. Also make sure that when a client is terminated (i.e. receives the InterruptedException), it unregisters itself from the server.

RunChat

Now create a class "RunChat" in programming3.chatsys.threads. The class has a static main method that will:

- Create a TextDatabase object.
- Create a ThreadServer with the TextDatabase object.
    - Create a thread with the server
    - Start the server's thread
- Create a first ThreadClient named "client1"
    - Create a thread for client1
    - Start the thread of client1
- Create a first ThreadClient named "client2"
    - Create a thread for client2
    - Start the thread of client2

Add logging information inside both the clients and the server to know what they are doing when you will execute your code.

Run your program and you should notice that the clients are registering with the server in parallel, sending a message and receiving it. Try running the program multiple times. You might notice that sometimes the server receives a message from both clients and sometimes only from one. We will discuss why this happens in the next lecture and see how we can solve this problem in the next exercise session.

Change the version number inside the pom.xml, commit your code with Git and create a tag "0.4.0".

ChatSys 0.4.1 – Refactoring

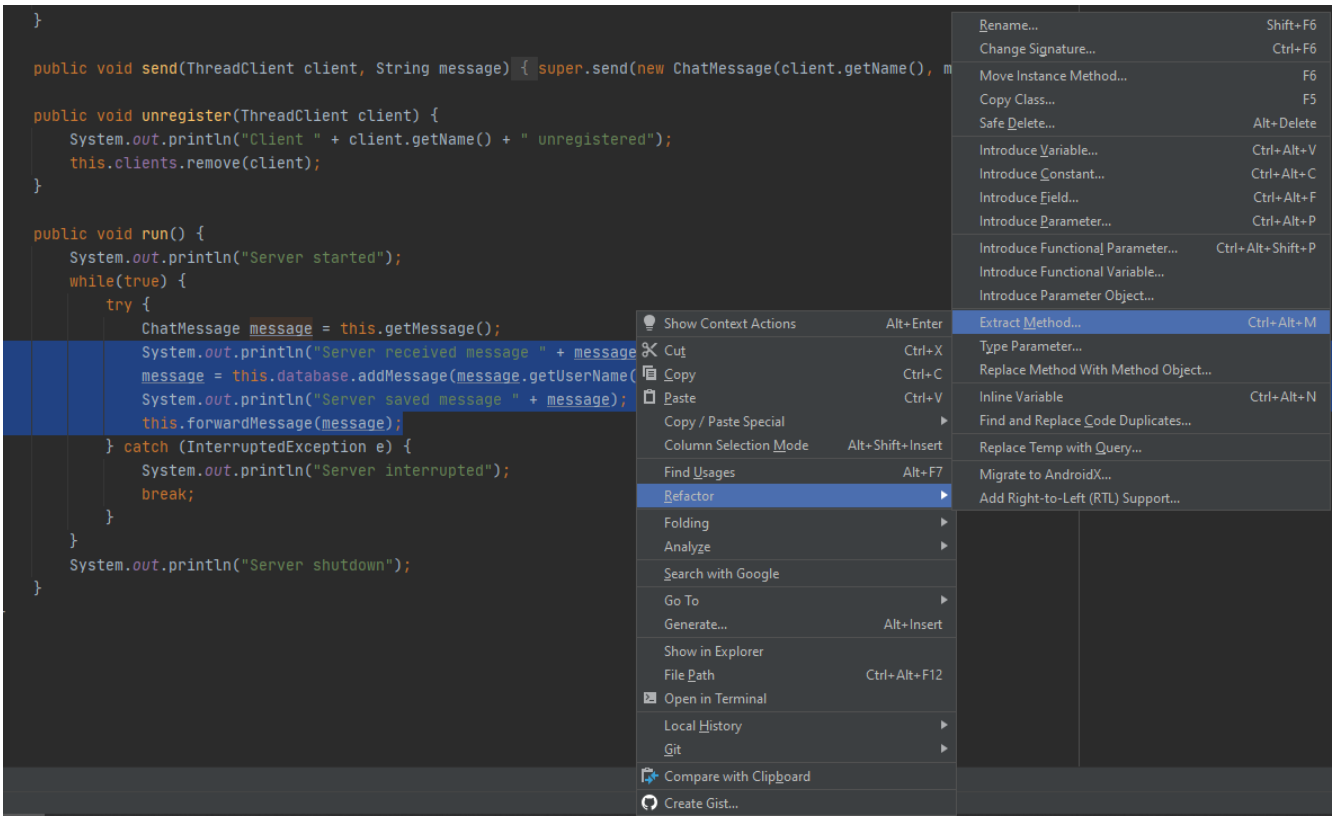
The goal of this task is to improve the structure of our current code base so future changes will be easier to make. The goal is also to make you learn how to use refactoring tools in your IDE. Refactoring is the process of changing your source code without altering its behavior in order to improve its structure and readability. Any good programmer should perform refactoring on a regular basis, otherwise the source code becomes more difficult to change and of lesser quality over time.

Look at the code of run in both the client and the server. You might find some similarity in as both the clients and the server work like this:

- Do some initialization
    - The client registers itself with the server and sends a message
    - The server doesn't do anything special, besides maybe logging some information to the console with System.out.println.
- Inside a loop, receive a message
    - If a message is received, do something with it
        - The client simply prints the message received
        - The server adds the message to the database and forward it
    - If the client or server is interrupted, break the loop
- After the loop has been broken, the client and server do something
    - The client unregisters itself with the server
    - The server do nothing (or print some logging information)

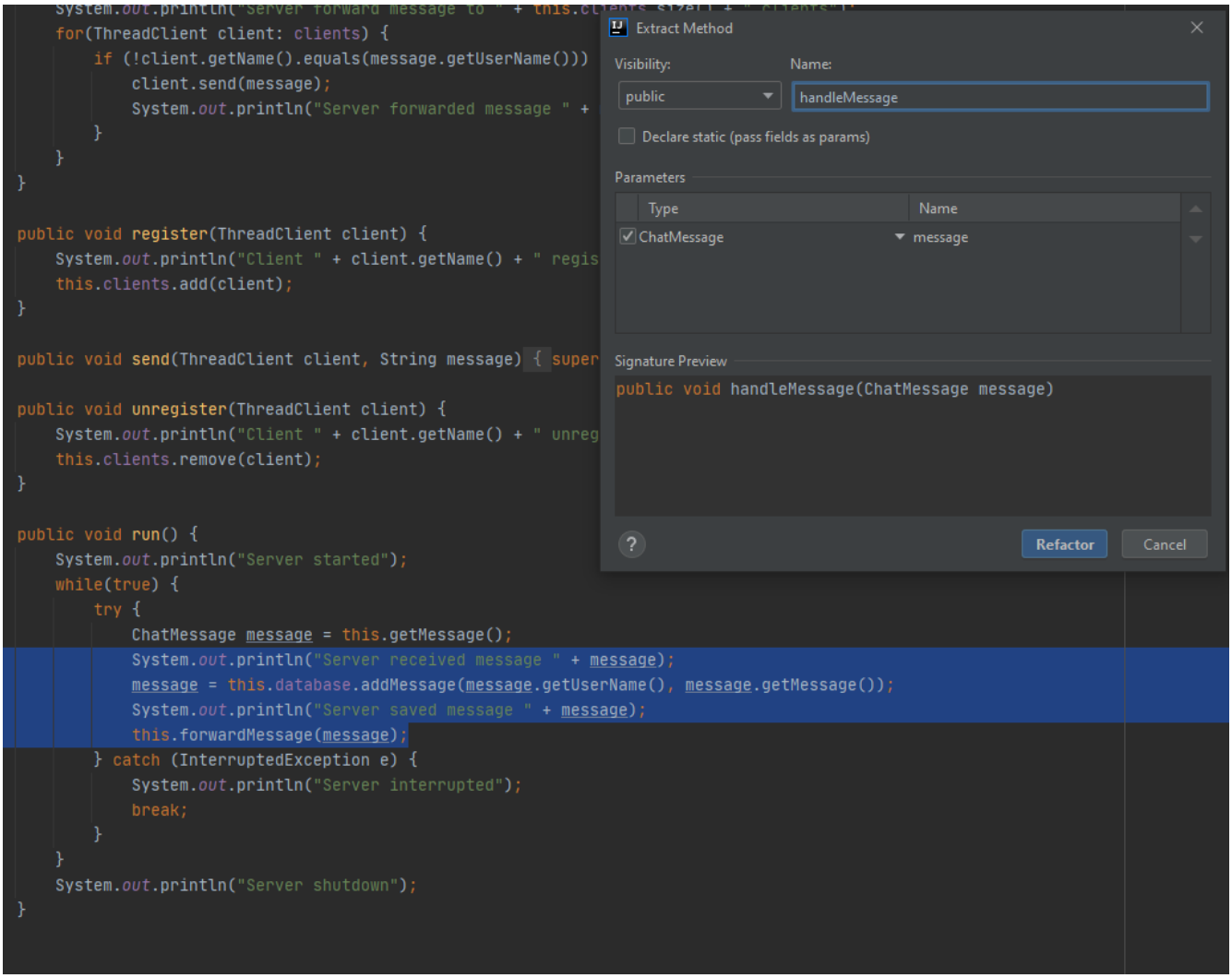
In order to avoid duplicate code, we will extract several methods from the run method of both the client and the server.

Inside a method, you can select several lines of code, right click on the selection and choose Refactor -> Extract Method.

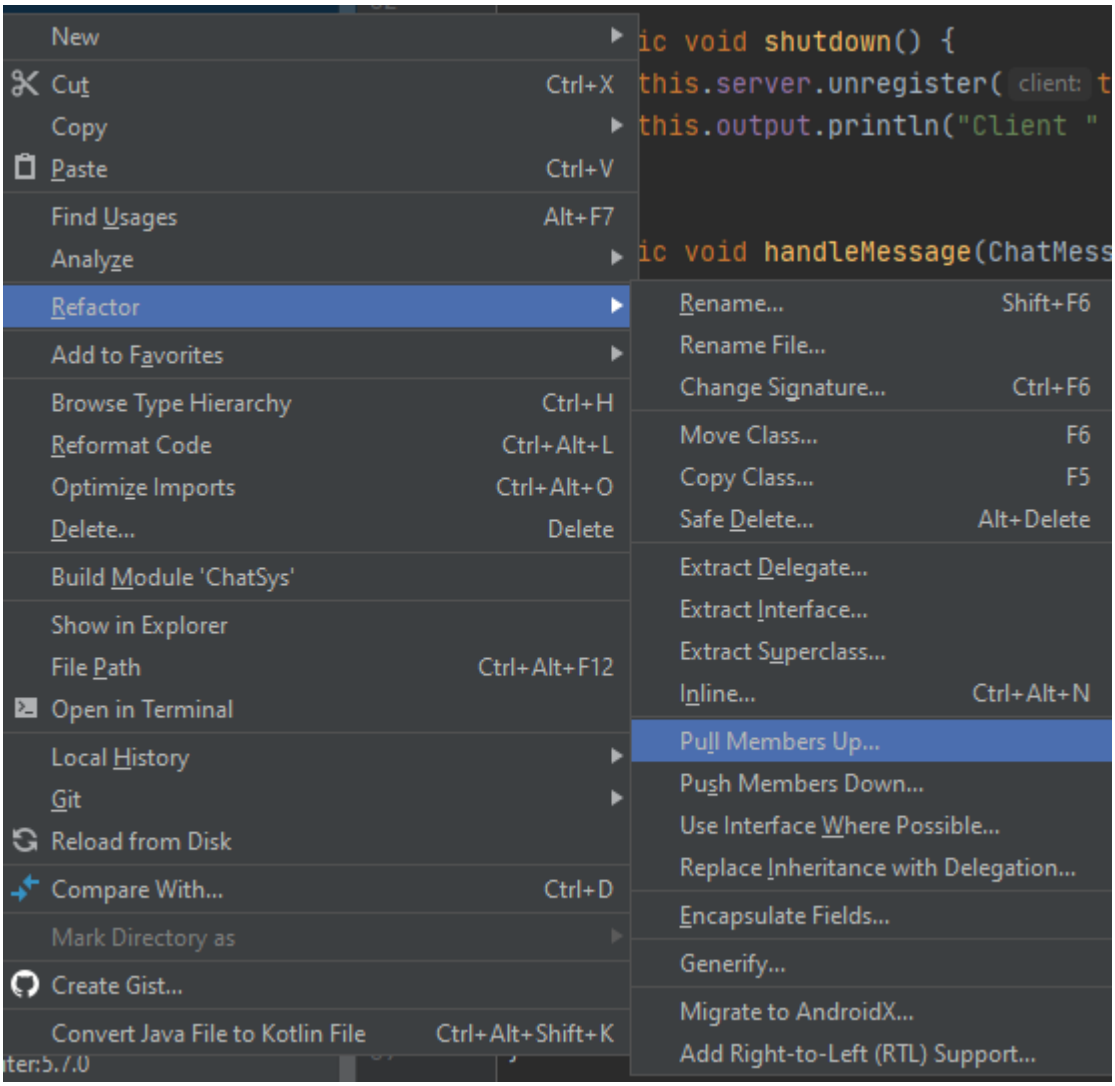


This will create a new method with the selected lines of code and replace the lines of code with a call to the new method.

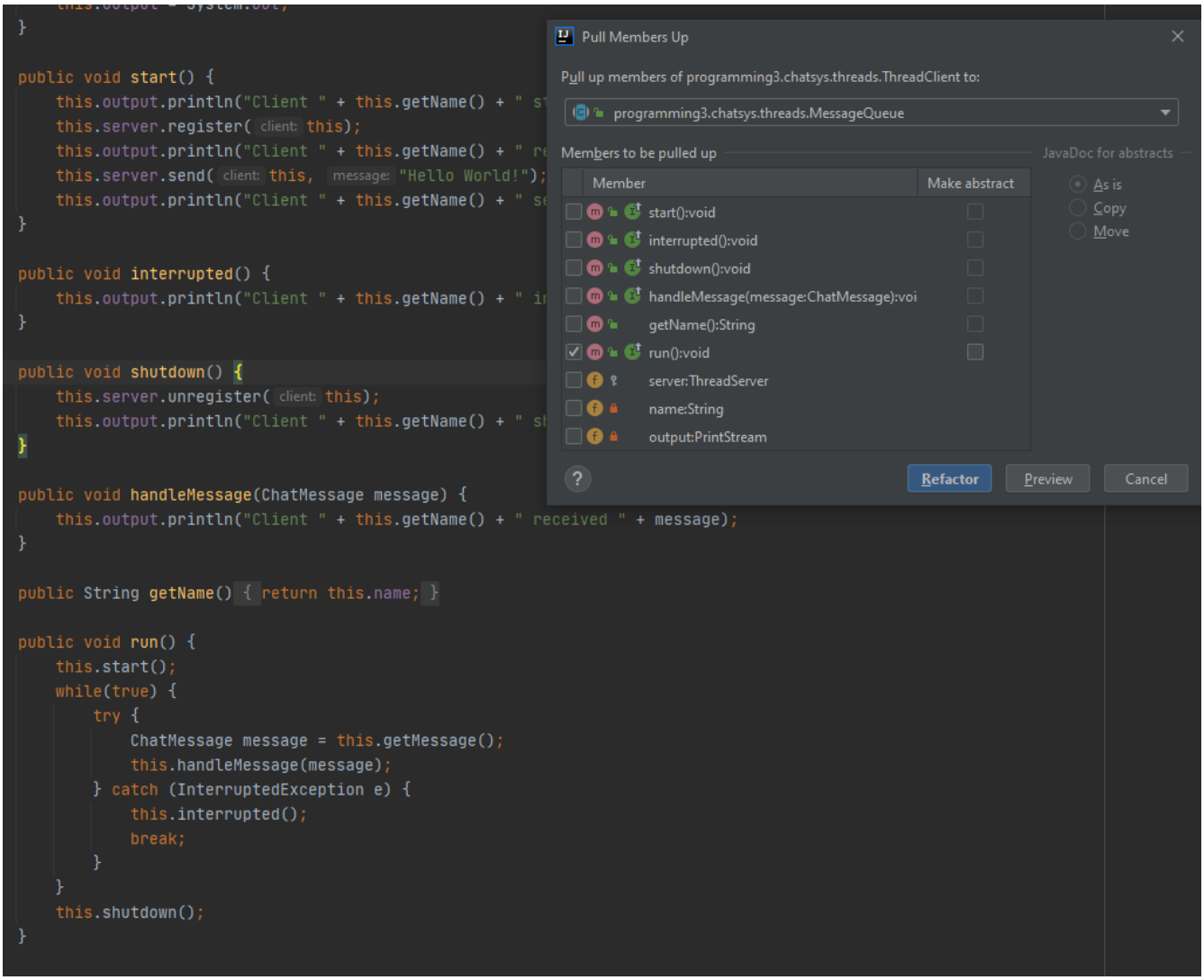
Inside both the client and the server, create methods “initialize”, “handleMessage” and “shutdown” by extracting a method from the client/server specific code described above. For example, extract the lines of code where the server adds a message to the database and forwards it to the clients in a method named “handleMessage”.



After extracting the 3 methods in both the client and the server, you should have the same code in their run method. Right click on one of the classes (e.g. ThreadClient), and choose Refactor -> Pull member up.



Choose the run method and move it to the superclass (MessageQueue).




```
}

public void start() {
    this.output.println("Client " + this.getName() + " st
    this.server.register( client: this);
    this.output.println("Client " + this.getName() + " re
    this.server.send( client: this,  message: "Hello World!");
    this.output.println("Client " + this.getName() + " se
}

public void interrupted() {
    this.output.println("Client " + this.getName() + " in
}

public void shutdown() {
    this.server.unregister( client: this);
    this.output.println("Client " + this.getName() + " sh
}

public void handleMessage(ChatMessage message) {
    this.output.println("Client " + this.getName() + " received " + message);
}

public String getName() { return this.name; }

public void run() {
    this.start();
    while(true) {
        try {
            ChatMessage message = this.getMessage();
            this.handleMessage(message);
        } catch (InterruptedException e) {
            this.interrupted();
            break;
        }
    }
    this.shutdown();
}
```

Make MessageQueue implement runnable and define 3 abstract methods: initialize, handleMessage and shutdown. You should now be able to delete the run method from the other class (e.g. ThreadServer). Also remove the “implements Runnable” in both ThreadClient and ThreadServer.

We now have a cleaner version of the client and the server where the run method is defined in MessageQueue and the different implementation of initialize, handleMessage and shutdown are defined inside the clients and the servers.

Don’t forget to update the version number, commit your code and tag it.

ChatSys 0.4.2 – Refactoring the Database

Let's do more refactoring, this time in relation to the Database.

TextDatabaseItem

In the previous exercise, we created two classes: ChatMessage and User. If you look at them, these two classes are relatively similar:

- They have a "format" and "parse" method. However, their implementation is different.
- They provide a method "save". Depending on how you implemented them, they might be very similar or even the same.

Having duplicate code in two different classes is usually not a good thing. If you must change one class, then it is likely you will have to update the implementation in the other class too. If you forget to do it, you might eventually run into some bugs. The best way to avoid duplicate code in Java is to make use of inheritance.

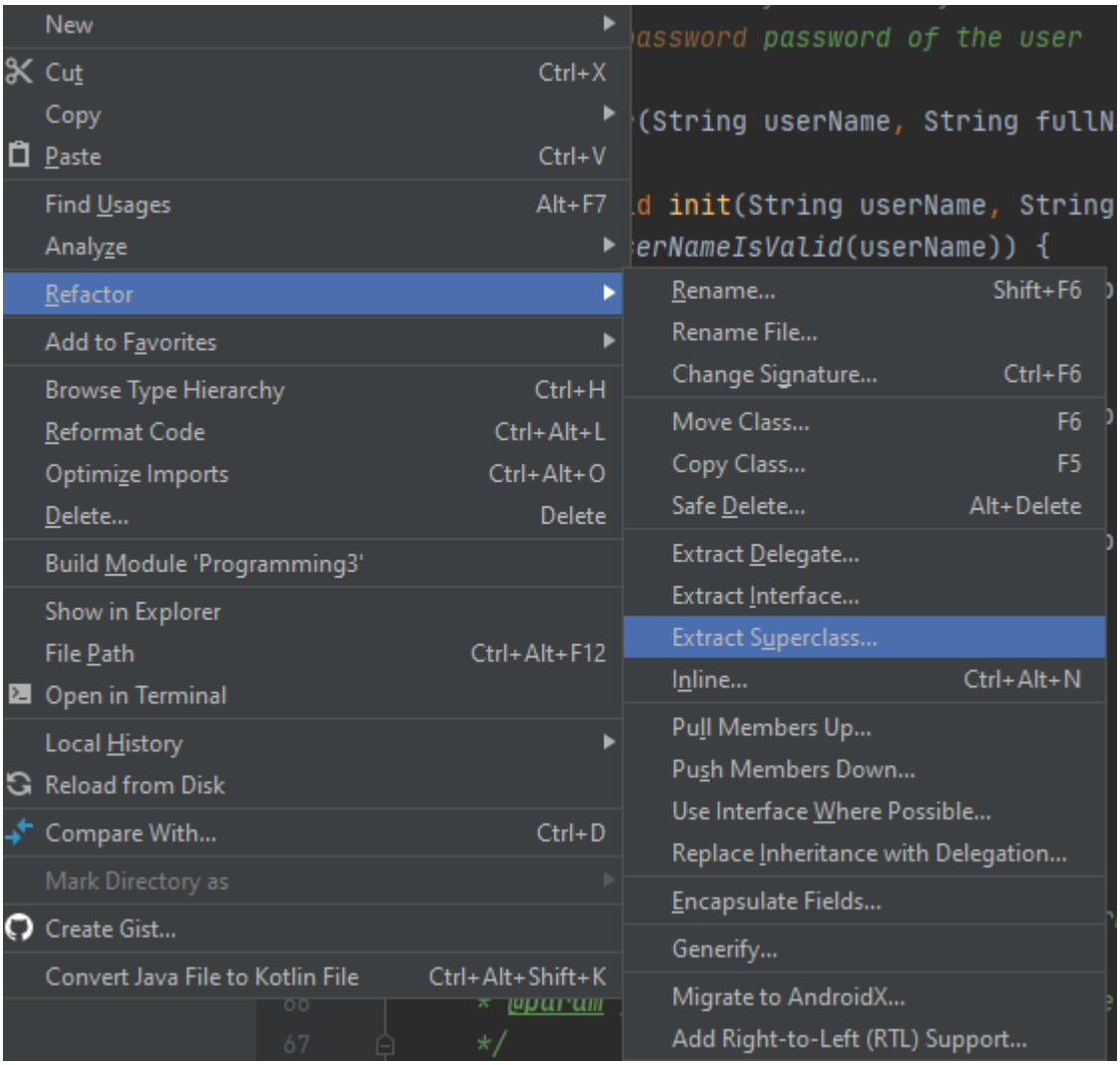
Ideally the save method of both ChatMessage and User should do the following:

- Open a BufferedWriter to the text file given as parameter
- Call the format method of either ChatMessage or User
- Write the output of format in the BufferedWriter
- Flush and close the BufferedWriter

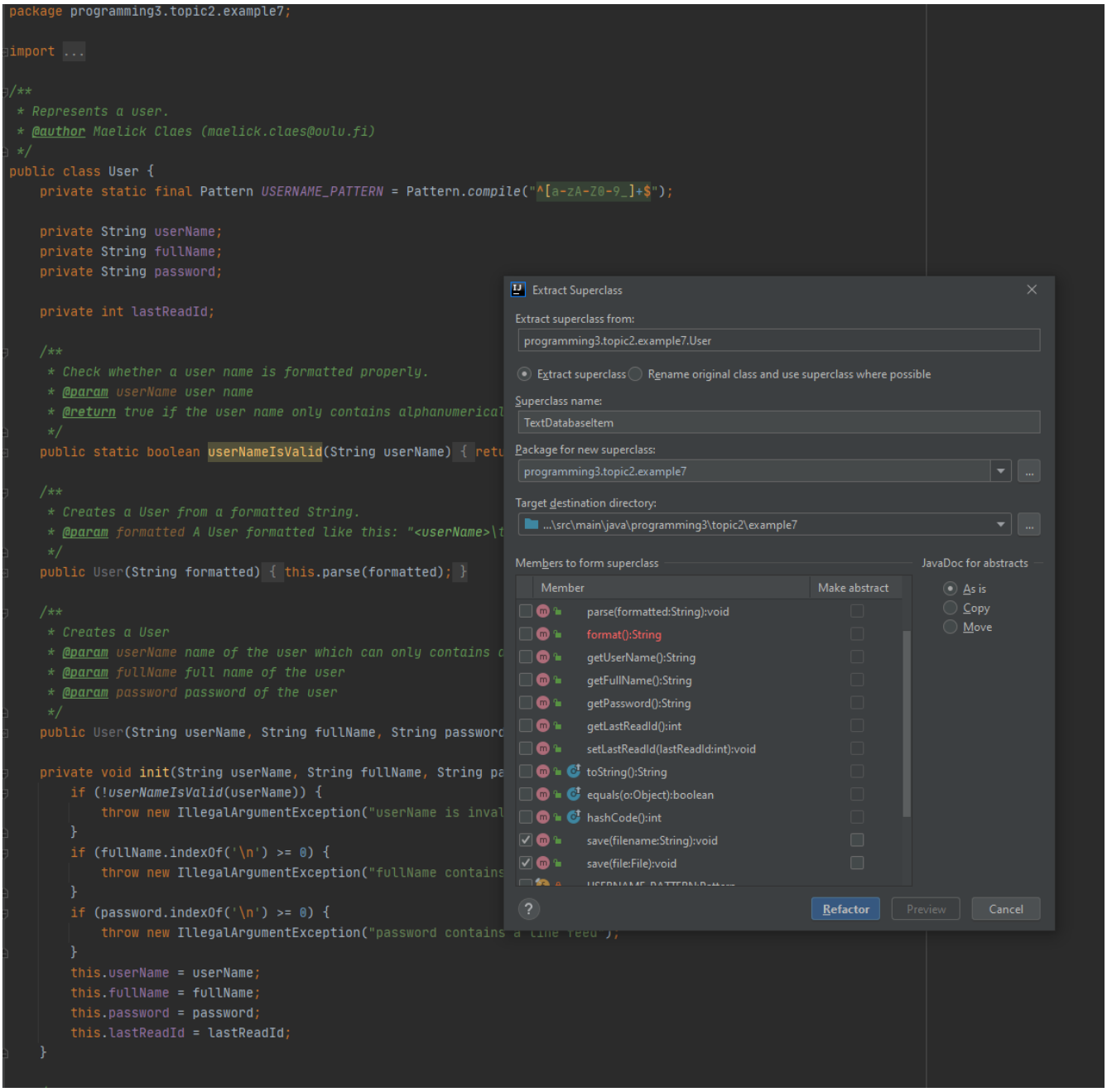
(If your implementation is different (e.g. you don't call the format method in the save method), it might be a good idea to update it to match the one detailed above.)

As we see, the only difference in their (ideal) implementation is the call to a method defined in each class. This is a good case for creating a super class for both ChatMessage and User, and have the save method defined in this super class.

In this case, we are going to create a new class that we will call "TextDatabaseItem" and that will provide a single implementation of save. For this, we can use the refactoring tools of IntelliJ. Right click on one of the two classes (e.g. User) and select Refactor -> Extract Superclass.



A window will open and will allow you to select which methods from the class opened you want to move to the superclass. Call the class “TextDatabaseItem”, select the method save and click Refactor.



Now User inherits from TextDatabaseItem and doesn't contain the save method anymore as it is defined in the new superclass. Now open ChatMessage, make it inherit from TextDatabaseItem and delete the save method.

Now if you open TextDatabaseItem, you will notice it doesn't compile. This is because the method save calls the format method which isn't defined in TextDatabaseItem. However, it is defined in both subclasses (User and ChatMessage). To solve this problem, we are going first to make TextDatabaseItem an abstract class. Then, we are going to create an abstract method format. For creating the method, we can right click on the call to the method format in save, and choose "Show Context Actions" (or press Alt+Enter). In the new menu that opens, select "Create abstract method …". Change the access of the newly created method from protected to public.

Database

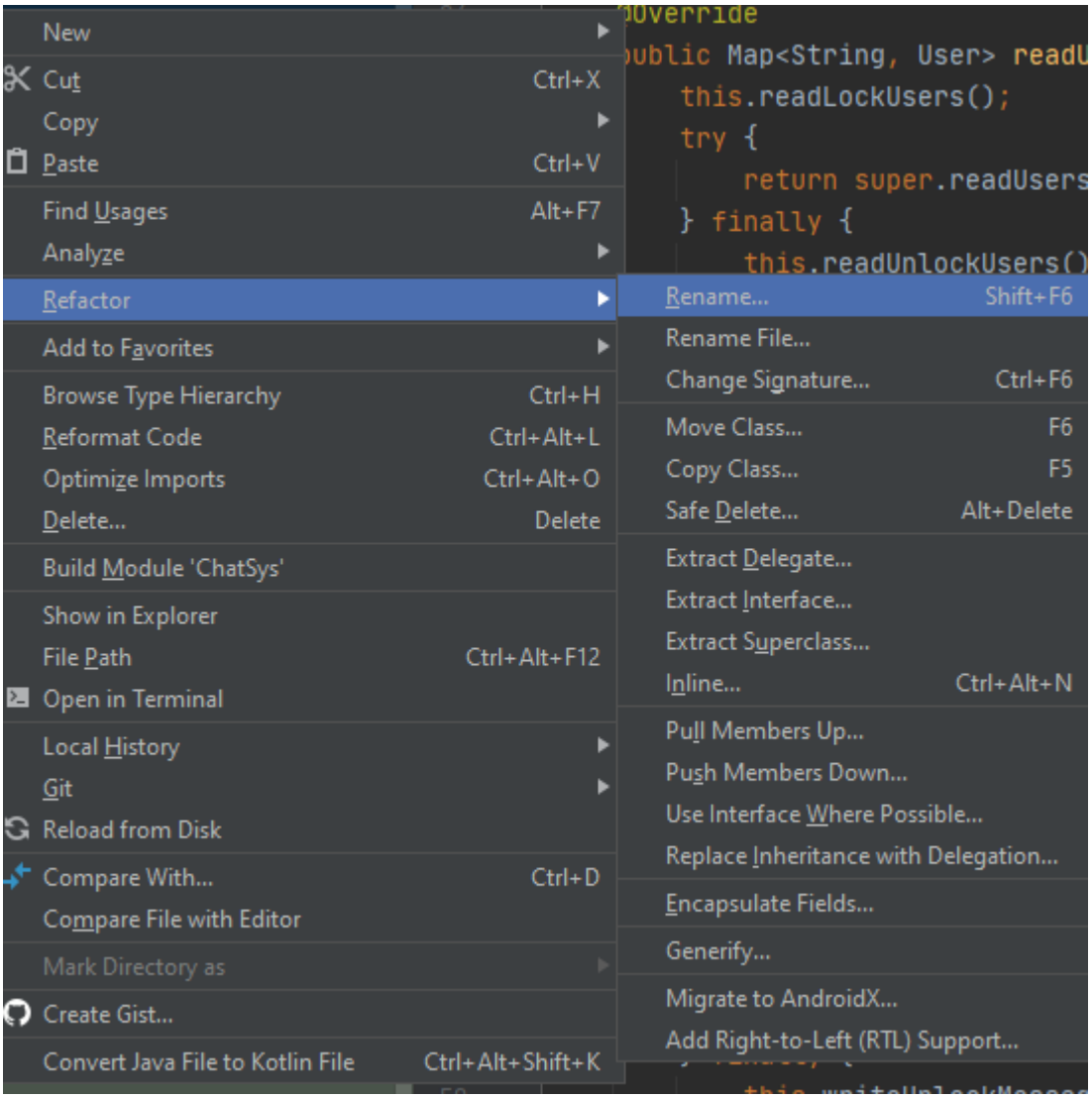
Now we are going to make some changes to the database. Right now we are using a simple implementation of a text database. However, in the next exercise session we will make changes to the implementation of this database. And later, we will create a new type of database relying on a real SQL database.

However, we don't want to lose our current implementation for the Database. Instead of having a single database implementation, it could be a good idea to have the possibility to decide which database we want to use. In order to make this possible, we are going to create an Interface that will define which database we use.

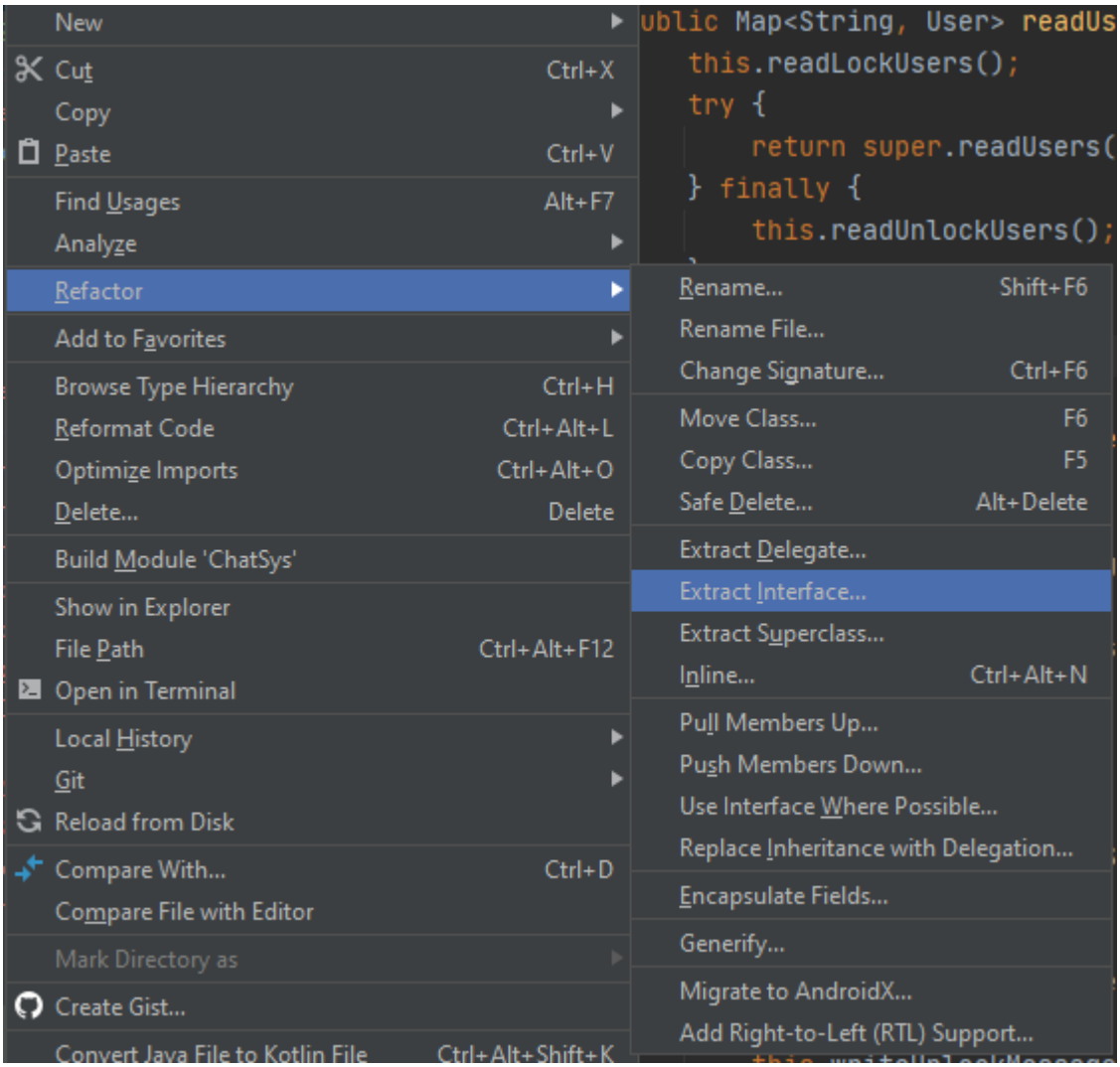
The interface should have the following methods:

- Map<String, User> readUsers(): Reads users from the database file.
- List<ChatMessage> readMessages(): Reads chat messages from the database file.
- ChatMessage addMessage(String userName, String message): Adds a ChatMessage to the database.
- boolean register(User user): Add a user to the database if it is not yet inside.
- List<ChatMessage> getUnreadMessages(String userName): Gets unread messages for a user and updates the user's last read id in the DB.
- boolean authenticate(String userName, String password): Check whether a pair of username and password are valid
- int lastId(): Returns the last id used in the message database.

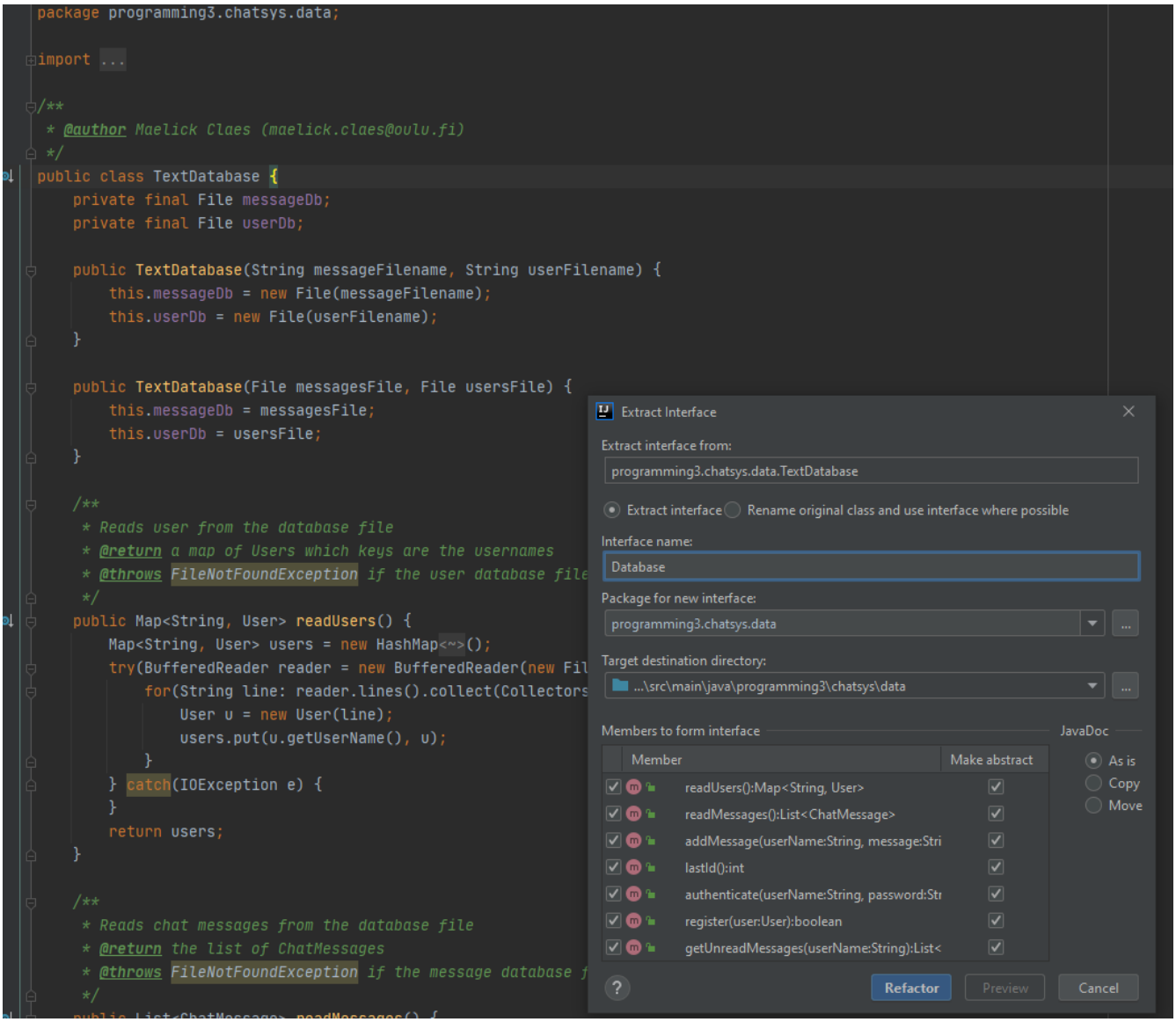
First let's start by renaming our Database class as "TextDatabase". For this, right click on the Database class in the project tab of IntelliJ -> Refactor -> Rename.



Rename the class as TextDatabase, then right click on the class again and go in Refactor -> Extract Interface.



You have a menu similar as the one for introducing a superclass. Select the methods that are already implemented in the TextDatabase. Name your interface “Database” and click Refactor.



Now add the missing method definitions in the Database interface, and implement them in TextDatabase.

Now we need to update the rest of the code. Right click on TextDatabase, and select “Find Usages”. This will show you every place where the TextDatabase is used. Every time a variable is declared of type TextDatabase in another class, you should change the type to Database. The only explicit references to TextDatabase should be inside your tests for TextDatabase, or when you call the constructor of TextDatabase. But **all other variables should be of type Database** and not TextDatabase.

When you are done, run all your unit tests to make sure your code is still working properly.

Don’t forget to update the version number, commit your code and tag it.

ChatSys 0.4.3 – AnsweringThreadClient

So far our clients only send messages when they register with the server. Now we would like to have some clients that can reply to a message when they receive one.

Because of the refactoring we did, making this change is quite straightforward.

Create a new class AnsweringThreadClient that inherits from ThreadClient.

Redefine the method handleMessage so that when the client receives a message:

- the client prints the message in the same way as ThreadClient do. This can be done by calling super.handleMessage(message).
- If the message is “Hello World!”, the client sends to the server a message “Hello <client>!” where <client> is replaced by the name of the client who sent the incoming message (message.getUserName()).

Update RunChat so that you create different types of clients: some simple ThreadClients that just send “Hello World” and some AnsweringClients that also reply to others message.

Test your code and see how it behaves.

Use an Executor to create many ThreadClients and AnsweringThreadClients. See how they behave.

With many clients, you might notice that the system doesn’t behave as expected as some messages are not received by the server. This is normal and we will see in the next lecture/exercise session how we can solve this problem.

Don’t forget to update the version number, commit your code and tag it.

ChatSys 0.4.4 – Thread-safe MessageQueue & ThreadServer

Note: only start this part of the exercise after the 3rd lecture (Monday 26th)

If you have run multiple times your threaded chat system, you should have noticed that not all messages are always received by the server. This occurs because multiple threads try to access to the same resource, the server's message queue, at the same time. If two clients try to write in the server's message queue at the same time, this can cause only one message being written in the queue!

In order to solve this, we can use a BlockingQueue instead of a Queue to store our messages. BlockingQueue is a superinterface of Queue, and the implementations provided by Java are thread safe, which means that two threads can access this class simultaneously without any problem.

Replace the queue of type MessageQueue with a BlockingQueue. You can use a LinkedBlockingQueue as implementation of BlockingQueue. We can also simplify and improve the implementation of MessageQueue. Replace the method getMessage(int waitTime) with a method getMessage() that simply calls the "take" method of the BlockingQueue and returns its value.

Then, in order to make our chat system fully thread-safe, we should also secure the server's set of clients. For that, simply call Collections.synchronizedSet in the constructor (e.g. clients = Collections.synchronizedSet(new HashSet<ThreadClient>());)

Now our server and clients should be fully functional.

Don't forget to update the version number, commit your code and tag it.

ChatSys 0.5 – Thread-safe Database

Note: only start this part of the exercise after the 3rd lecture (Monday 26th)

It is not only variables in memory that can be problematic when working with multiple threads. Any resources that is accessed by multiple threads can potentially create race conditions. Files are such resources. Thus, there can be potential problems when different threads access the TextDatabase.

In our simple threaded chat application, the database is only accessed by the server. Thus, there is only one thread that access it because the server has only one thread. However, soon, we will implement a real network server which will be able to have multiple clients connection at the same time, using different threads. Thus, we are now going to end this exercise by securing the TextDatabase.

We can secure the database by creating two locks: one for the file storing the Users, and one for the file storing the ChatMessages. Every time the TextDatabase try to access one of these files for reading and writing, we must lock the appropriate lock, and once we are done unlock it. While we could use two simple ReentrantLocks, we are instead going to use ReentrantReaderWriterLock.

Implement those locks in a new class SecureTextDatabase that inherits TextDatabase and reuse as much code as possible (i.e. try to avoid duplicate code). Once you are done, don't forget to update the version number, commit your code and tag it.