INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

DEPARTAMENTO DE CIENCIAS E INGENIERÍA

Inventory Control with Java Concurrency

Alberto Castañeda Arana A01250647

Programming Languages Final Project

Querétaro, QRO **November 2021**

Abstract

For retail and manufacturing businesses it is extremely important to maintain consistency on their data so the real stock that the business has is equal to what they would expect from their sales and purchases. Managing an inventory by hand can lead to human error, which is where inventory control systems come to solve the problem. In this document we showcase an approach to solve the previous problem with a lightweight software application using concurrency and a simple user interface using Java. Along this document, we will go into detail with the problem presented, explain the programming language and paradigms used for this approach, and show the results from the implementation.

Content

Chapter 1
Context of the problem

1.1 History

Before the Industrial Revolution, merchants had to write down all of the products sold and bought every day by hand. This was incredibly inefficient and inaccurate, since this didn't account for stolen goods or human error. It was also a task that took a lot of time. It was not until the 1930s that Harvard University created a punch card system for business. This machine was able to record inventory and sales data based on punch cards customers would fill out for items. However, this didn't become widely used because of the machine's high cost and slow processing. As computers became more efficient and cheaper, newer solutions came to the market. During the 1990's, companies began experimenting with inventory management software to be able to record income and outcomes of their products. Spreadsheet applications also started to rise, with Microsoft Excel launching in 1985 and becoming a new alternative for inventory management.

1.2 State of the Art

While there is no definitive tool or method to manage an inventory, there are some that are commonly seen, each having their own advantages and disadvantages. The following are a couple of examples:

	Advantages	Disadvantages
Manual (By hand)	● Intuitive ● No need for any hardware or connection	● Most susceptible to human error ● Time consuming ● Errors may be hard to track or fix
Excel	● Easy to use ● Version control for changes ● Easy to analyze statistics	● Hosted on a single computer, susceptible to file loss. ● Requires manual modification of data
Inventory Software	● Software may feature inventory modification remotely	● Might need manual adjustment in case of stock theft or external

	● Allows multiple concurrent access to the inventory. ● Software may feature an audit log to visualize transactions.	factors ● The most expensive alternative. ● Hard to use for non-tech savvy people.

While an inventory software might sound like the straightforward go to solution for a business that manages a big sized inventory and/or has a lot of client petitions, SaaS (Software as a Service) of this kind are usually expensive for small businesses. A selection of existing inventory management software monthly prices resulted in an average of $1,1225.00 dollars per month. While inventory management software can avoid huge financial losses by protecting from accounting errors, many small businesses can find SaaS licenses to be too expensive. There is a need for an affordable, secure, and friendly inventory management software. For this project, we will create a simple lightweight solution that allows users to handle their inventory with multiple possible clients.

Chapter 2
Solution

2.1 Constraints

One thing we must have in mind in order to create an inventory management software is how transactions will be made. In practice, one provider can make multiple requests simultaneously to other providers. An inventory product's stock can increase or decrease atomically, so it must always be consistent with the inputs and outputs. On the other hand, a product order must handle any case where there is not enough existence for the requested product. To tackle this problem with a programming mindset, the inventory's data must be stored somewhere with a middle interface that controls this inventory with multiple client requests. In order to do this, we must understand the Concurrency programming paradigm.

2.2 Concurrency

Concurrency in computer science means multiple computations happening at the same time, and potentially interacting with each other without affecting the final outcome. The computations may be executed on multiple cores on the same chip, time-shared threads on the same processor, or executed on physically separated processors. Concurrency is often confused with the Parallel computing paradigm. While they might look similar, they are rather different in reality. Parallel computing has a deterministic control flow, while concurrent computing doesn't, and Concurrent Computing can be done on a single CPU. In other words, parallel computing is about speeding up tasks, while concurrent computing reduces or hides latency.

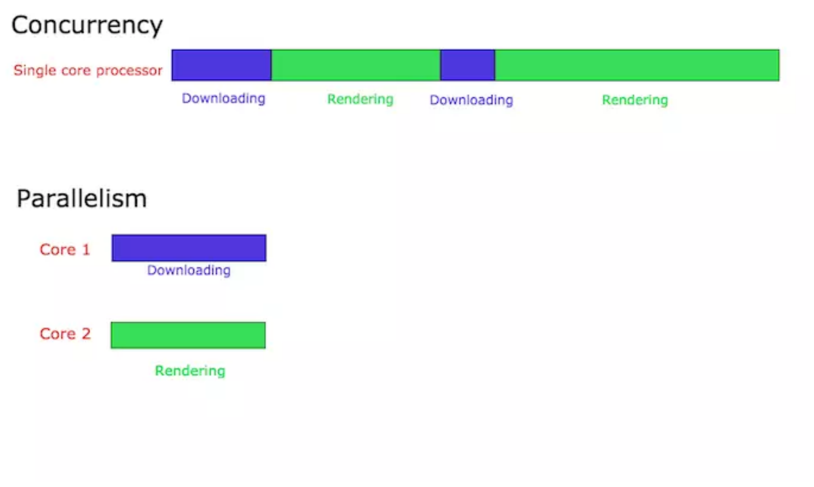


Image 1. Concurrency vs Parallelism

2.3 Architecture

In order to allow multiple simultaneous inventory transactions and operations in the context of our inventory application we must design our application's architecture with multiple concurrent operations in mind, which is where the concurrency paradigm becomes helpful.

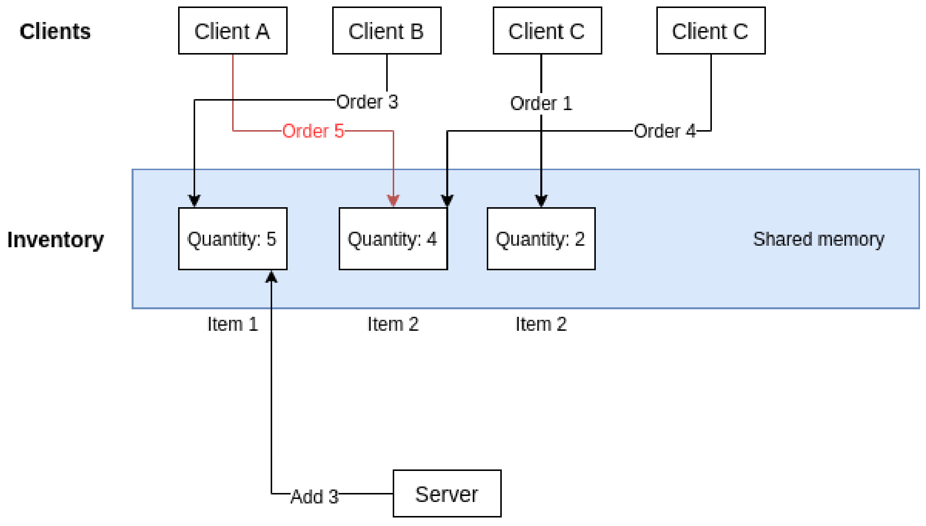


Image 2. Inventory management with multiple clients and one server sharing memory. Red arrows marked as failure requests.

The approach chosen for this project was to handle each client connection as a separate thread with every connection being established via Sockets. Sockets are used to represent the connection between a client program and a server program, in our case between the inventory server and the multiple clients. Sockets provide a reliable, point-to-point communication channel over TCP.

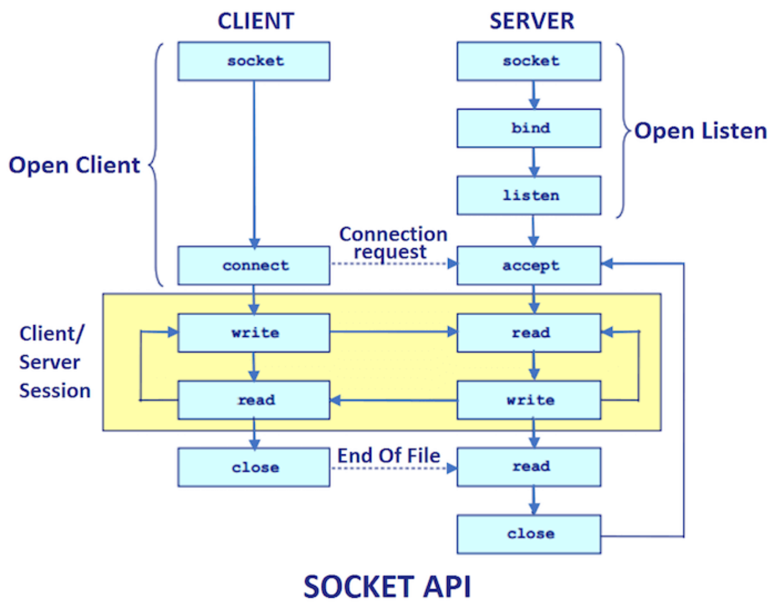


Image 3. Socket Connection Architecture

By instantiating each socket as a different thread in our server instance, we are able to handle multiple independent requests using one shared memory. The following would be our application's general architecture as a result.

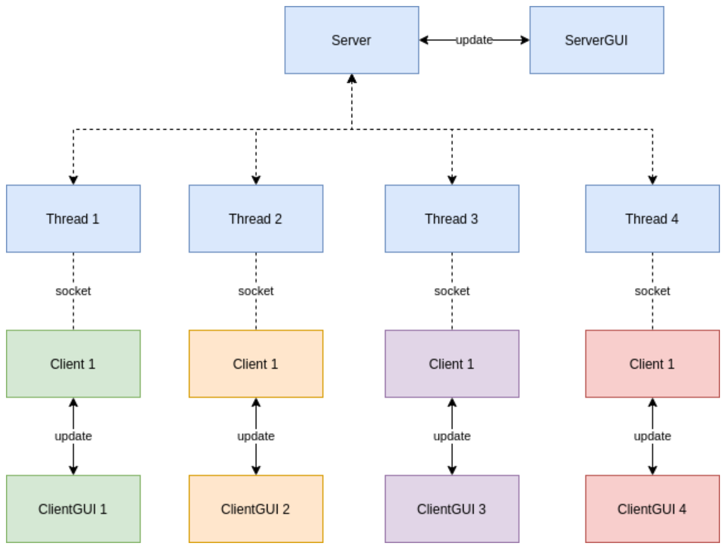


Image 4. Threads per client communicating with the server via sockets

2.4 Programming Language

Design specifications aside, the language chosen for this project was Java. The reason it was chosen is that Java provides out-of-the-box thread management and creation, as well as socket classes for interconnection between multiple instances. Additionally, Java is included with the Swing toolkit which will provide us with tools to create a graphical interface to improve the applications usability.

2.5 Implementation

The application can be divided in two parts: the server application which will store the inventory and process requests, and the client who will create requests to the server and consume the server inventory's data. That being said, the lifetime of our sub-applications once instantiated can be represented by the following:

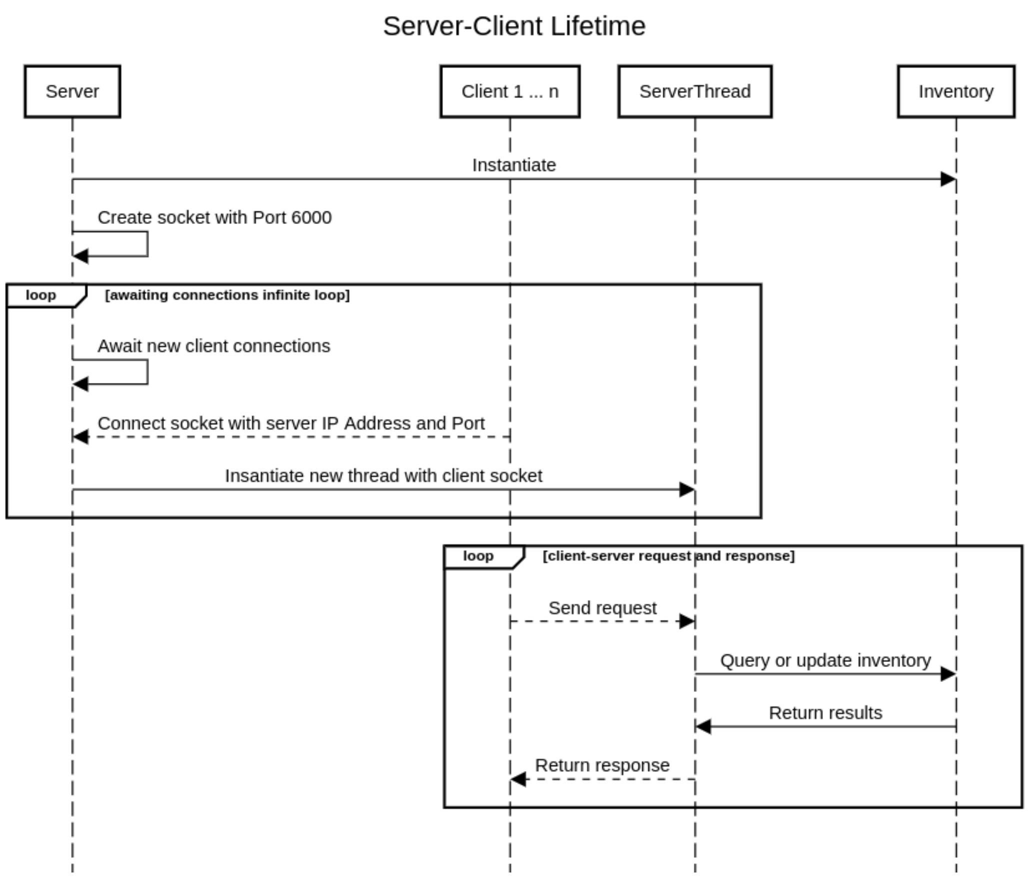

Server-Client Lifetime
Server
Client 1 ... n
ServerThread
Inventory
Instantiate
Create socket with Port 6000
loop
[awaiting connections infinite loop]
Await new client connections
Connect socket with server IP Address and Port
Insantiate new thread with client socket
loop
[client-server request and response]
Send request
Query or update inventory
Return results
Return response

Image 5. Server-client lifetime sequence diagram

```
class Server {
   private Socket socket;
   private ServerSocket server;
   protected Inventory inventory;

   private final int PORT = 6000;

   void init() {
      try {
         server = new ServerSocket(PORT);
      } catch (IOException error) {
         error.printStackTrace();
      }
      System.out.println("Server started");
      System.out.println("Waiting for clients...");

      while(true) {
         try {
            socket = server.accept();
            System.out.println("Server - Connection establilshed.");
        } catch (IOException error) {
            System.out.println(error);
        }
        new ServerThread(this, socket).start();
      }
   }
}
```

Code Snippet 1. Creating server socket and waiting for new client connections for new threads.

```
class Client {
   private Socket socket;
   private Scanner in;
   private PrintWriter out;
   private Inventory inventory;

   public Client(String address) {
      try {
         int PORT = 6000;
         socket = new Socket(address, PORT);
         System.out.println("Client - Connection established.");

         in = new Scanner(socket.getInputStream());
         out = new PrintWriter(socket.getOutputStream(), true);

         while (true) {
           String response = in.nextLine();
           System.out.println("Response from server: " + response);
           processResponse(response);
         }
      } catch (IOException error) {
         System.out.println(error);
      }
   }
}
```

Code Snippet 2. Connecting to server socket from client

Both the client and server classes store their counterpart socket. These sockets contain an input stream and an output stream. The former is what is received from their counterpart, while the latter is what is sent. For example, in our client the output stream would be equivalent to what is sent to the server while the input stream is what is received from the server. The Scanner will listen on the input stream, and wait for a line to be sent. This is equivalent to a program waiting for an input to be typed in order to proceed. However, the input this time being the input stream received instead of a manual input.

We also store an instance of the Inventory class on the server class and the client class. The Server inventory could be considered the "real" inventory, while the client inventories may be out of date and only a snapshot of the real inventory. In practice, our inventory's data should be stored in a robust database management system like MySQL or Postgres and our server instance would be the middle interface that joins clients and the database. For the simplicity of this project we will be storing the inventory data on the host's memory as an array of Record models. Record models will store our inventory's data relevant information, like the item's name, price, and the item's available stock.

Once our connections between server and clients are established, we can now initiate making requests as a client to the server. Going back to the previously mentioned output stream and input stream, it is important to note that these messages are sent via string messages. We know that there will be different types of requests that a client will need to do, like a list request or an order request for example. That being said, in order to distinguish what request is being sent we must implement a serialization mechanism. Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file in order to recreate it back when needed. This recreation process is called deserialization. One of the most common serialization techniques is JSON serialization. JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for both humans and machines to parse and generate. We can pass multiple variables needed to a new JSON object and convert this object to a string to send it to the output stream, which can then be deserialized to get the original values sent.

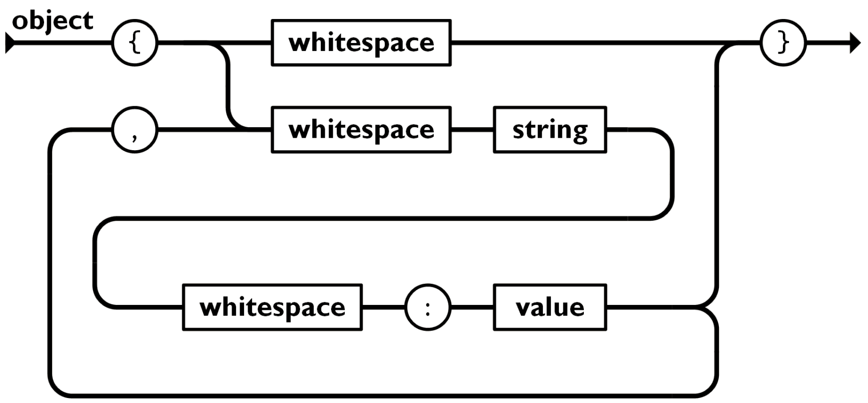


Image 5. JSON syntax structure

Unfortunately, JSON is not a standard library included in Java which is why I imported the public org.json library from this repository.

With our new serialization communication technique, we can now start making different kinds of requests from clients and make an appropriate response from the server side.

```
class Client {
   ...
   public void fetchRecords() {
      JSONObject object = new JSONObject();
      object.put("Request", Constants.LIST_ACTION);
      out.println(object);
      // Wait for fetch response
      String response = in.nextLine();
      processResponse(response);
   }


   public void orderRecord(int id, int quantity) {
      JSONObject object = new JSONObject();
      object.put("Request", Constants.ORDER_ITEM);
      JSONObject itemObj = new JSONObject();
      itemObj.put("id", id);
      itemObj.put("quantity", quantity);

      object.put("item", itemObj);
      out.println(object);
   }

}
```

Code Snippet 3. Client functions to create request jsons. First function being a simple list request to get records. The second function sends an order request and requires the id and quantity of the product requested.

```
class ServerThread {
    ...

    private String processRequest(String request) {
    JSONObject deserialized = new JSONObject(request);
    String requestType = (String) deserialized.get("Request");
    JSONObject response = new JSONObject();
    response.put("Request", requestType);
    switch (requestType) {
        case Constants.LIST_ACTION -> {
            Record[] records = server.getRecords();
            JSONArray array = new JSONArray();
            for (Record record : records) {
                JSONObject obj = new JSONObject();
                obj.put("id", record.item.id);
                obj.put("name", record.item.name);
                obj.put("price", record.item.price);
                obj.put("quantity", record.quantity);
                array.put(obj);
            }
            response.put("records", array);
            server.addLog("List request from: " + socket.getInetAddress().getHostName());
            return response.toString();
        }
        case Constants.ORDER_ITEM -> {
            JSONObject itemObj = (JSONObject) deserialized.get("item");
            int id = itemObj.getInt("id");
            int quantity = itemObj.getInt("quantity");
            server.addLog("Order request ID: " + id + " Quantity: " + quantity + " from: " +
socket.getInetAddress().getHostName());

            boolean success = server.order(id, quantity);
            if (success) {
                Record[] records = server.getRecords();
                JSONArray array = new JSONArray();
                for (Record record : records) {
                    JSONObject obj = new JSONObject();
                    obj.put("id", record.item.id);
                    obj.put("name", record.item.name);
                    obj.put("price", record.item.price);
                    obj.put("quantity", record.quantity);
                    array.put(obj);
                }
                response.put("records", array);
                server.refresh(server.getRecords());
                return response.toString();
            } else {
                JSONObject error = new JSONObject();
                error.put("Request" , Constants.ITEM_UNAVAILABLE);
                return error.toString();
            }
        }
        default -> {
            System.out.println("Unknown request type");
            return "Unknown request";
        }
    }
  }
}
```

Code Snippet 4. Server processing function. Obtains the request parameter of the json and conditionally processes it. Returns a response JSON to be processed similarly by the client.

```
class Client {
     ...
   protected void processResponse(String response) {
      JSONObject deserialized = new JSONObject(response);
      String request = (String) deserialized.get("Request");
      switch (request) {
         case Constants.LIST_ACTION:
         case Constants.ORDER_ITEM: {
            JSONArray records = deserialized.getJSONArray("records");
            Record[] results = new Record[records.length()];
            for (int i = 0; i < records.length(); i++) {
               JSONObject json = records.getJSONObject(i);
               int id = (Integer) json.get("id");
               String name = (String) json.get("name");
               float price = ((BigDecimal) json.get("price")).floatValue();
               int quantity = (Integer) json.get("quantity");

               Item item = new Item(id, name, price);
               results[i] = new Record(item, quantity);
            }
            inventory = new Inventory(false);
            inventory.setRecords(results);
            refreshGUI(inventory.getRecords());
            break;
         }
         case Constants.ITEM_UNAVAILABLE:{
            onOrderUnavailable();
         }
         default: {
            break;
         }
      }
   }
```

Code Snippet 5. Client response processing function. Obtains the response obtained from a request to the server and does something with it. For example, our list and order action will both return an updated array of the inventory records.

One problem that might arise when using a shared memory in a concurrent program is concurrency related problems. This happens when multiple instances are attempting to read or modify the same shared memory location. Take for example a bank account that has $300 dlls in their account, where client 1 withdraws $300 and client 2 withdraws $200 from the same account simultaneously. Concurrent problems that could arise from this scenario are deadlocks for example. Deadlocks are a situation in which two computer programs sharing the same resource are preventing each other from accessing the resource in our case the bank account total. Moving on from this hypothetical situation to our inventory application, we want to protect our inventory shared memory from having concurrency problems since we know that multiple clients could potentially operate over the same item. Thankfully, we can use Java's synchronized functions for these inventory management functions. Synchronized functions are one of Java's two basic synchronization idioms (the other being synchronized statements). Making a function synchronized means that it is not possible for two invocations of the function on the same object to interleave. If a thread calls a synchronized function, other threads that also call it must wait for the original thread to end the function execution. By just doing this, we can avoid concurrency problems in our shared memory inventory.

```
class Inventory {
      ...
   public synchronized boolean orderRecord(int id, int quantity) {
      int originalQuantity = records[id].quantity;
      records[id].quantity = records[id].quantity - quantity;

      if (records[id].quantity < 0) {
         System.out.println("Not enough inventory for this item request");
         records[id].quantity = originalQuantity;
         return false;
      } else {
         System.out.println("Ordered succesfully");
         return true;
      }
   }

   public synchronized void addRecord(int id, int quantity) {
      for (Record record : records) {
         if (record.item.id == id) {
            record.quantity += quantity;
            System.out.println("Added item succesfully");
            return;
         }
      }
      System.out.println("Item not found");
   }
 }
```

Code Snippet 6. Inventory functions with synchronized keyword.

At this point, we could treat this project as done and handle our inventory via the command line. However, in order to make it more friendly it is time to add the graphical components for the application. As initially said, we will be using the Java Swing toolkit to create our graphical interface where the buttons and inputs will call our previously defined request functions. There is one caveat to constraining our application to a graphical interface though. If we wanted to test our application we would have to open multiple interfaces and control each of them. Testing that our application manages concurrently properly while using multiple windows would be complicated. In order to solve this problem, we can use the beauty of abstractionism in object oriented programming. If we create a super class with all of our GUI-independent functions and variables, while defining our GUI-dependent functions and mechanisms as abstract, we could create a GUI version and a CLI (Command line interface) version of both the client and server web applications. In the CLI subclass of the abstract class, we would just empty out the GUI functions or replace it with another functionality. The previous is a practice commonly used in software testing called "stubbing".

```
abstract class BaseClient {
   ...

   abstract void initializeGUI(Record[] records);

   abstract void refreshGUI(Record[] records);

   abstract void onOrderUnavailable();

   abstract void run();
}

public class Client extends BaseClient {
   private ClientGUI gui;

   ...

   @Override
   void initializeGUI(Record[] records) {
      gui = new ClientGUI(this);
      gui.init(records);
      gui.frame.addWindowListener(new WindowAdapter() {
         @Override
         public void windowClosing(WindowEvent e) {
            try {
               socket.close();
            } catch (IOException ex) {
               ex.printStackTrace();
            }
            System.exit(0);
         }
      });
   }

   @Override
   void refreshGUI(Record[] records) {
      if (gui != null) {
         gui.refresh(records);
      }
   }

   @Override
   void onOrderUnavailable() {
      gui.activateNoExistanceDialog();
   }

   @Override
   void run() {
        while (true){
            String response = in.nextLine();
            System.out.println("Response from server: " + response);
            processResponse(response);
        }
    }
}

class TestClient extends BaseClient {
    ...

    @Override
    void initializeGUI(Record[] records) {
        // Do nothing
    }

    @Override
    void refreshGUI(Record[] records) {
        // Do nothing
    }

    @Override
    void onOrderUnavailable() {
        // Do nothing
    }

    @Override
    void run() {
        while (true) {
            try {
                int randomID = (int) ((Math.random() * (4)));
                int randomQuantity = (int) ((Math.random() * (2 - 1)) + 1);

                System.out.println("Ordering " + randomID + " quantity " + randomQuantity);
                orderRecord(randomID, randomQuantity);
                String response = in.nextLine();
                processResponse(response);

                thread.sleep((int) ((Math.random() * (6000 - 3000)) + 3000));
            } catch (InterruptedException error) {
                error.printStackTrace();
            }
        }
    }
}
```

Code Snippet 7. Base abstract client class, GUI client subclass, and Test CLI client sublclass in order.

Having this new polymorphism of classes, we can now create our tests for the application. For our tests, we will want to create a server instance and multiple client instances with different threads that make random order petitions at random times. Java threads allow us to use the function *sleep* which makes the thread hang or wait for a specified time in milliseconds. Having multiple simultaneous requests on potentially the same items and logging those operations, we can see in the output that there is no error or unusual behaviour on our inventory management. The server subclass was modified so in case there is a concurrency error and somehow the stock quantity of an item becomes negative, it exits the whole program.

```
import java.util.ArrayList;

class TestClient extends BaseClient {
    private Thread thread;
    public TestClient(String address, Thread thread) {
        super(address);
    }

    @Override
    void initializeGUI(Record[] records) {
        // Do nothing
    }

    @Override
    void refreshGUI(Record[] records) {
        // Do nothing
    }

    @Override
    void onOrderUnavailable() {
        // Do nothing
    }

    @Override
    void run() {
        while (true) {
            try {
                int randomID = (int) ((Math.random() * (4)));
                int randomQuantity = (int) ((Math.random() * (2 - 1)) + 1);

                System.out.println("Ordering " + randomID + " quantity " + randomQuantity);
                orderRecord(randomID, randomQuantity);
                String response = in.nextLine();
                processResponse(response);

                thread.sleep((int) ((Math.random() * (6000 - 3000)) + 3000));
            } catch (InterruptedException error) {
                error.printStackTrace();
            }
        }
    }
}

class TestClientThread extends Thread {
    public void run() {
        TestClient client = new TestClient("localhost", this);
    }
}

class TestServer extends BaseServer {
    @Override
    void addLog(String log) {
        System.out.println(log);
    }

    @Override
    void refresh(Record[] records) {
        // Do nothing
    }

    @Override
    public boolean order(int id, int quantity) {
        boolean success = inventory.orderRecord(id, quantity);
        Record[] records = inventory.getRecords();
        for(Record record : records){
            System.out.println(record.item.name + " - " + record.quantity);
        }
        if(inventory.getRecord(id).quantity < 0) {
            System.out.println("CONCURRENCY ERROR");
            System.exit(1);
        }
        System.out.println("\n");

        return success;
    }
}


class TestServerThread extends Thread {
    public void run() {
        TestServer server = new TestServer();
        server.inventory = new Inventory(true);
        TestServerAdderThread adder = new TestServerAdderThread(server);
        adder.start();
        server.init();
    }
}

class TestServerAdderThread extends Thread {
    private TestServer server;
    public TestServerAdderThread(TestServer server){
        this.server = server;
    }

    public void run() {
        while (true) {
            if (server.inventory == null) continue;
             int randomID = (int) ((Math.random() * (3)));
            int randomQuantity = (int) ((Math.random() * (5 - 1)) + 1);
            System.out.println("Adding product " + randomID + " Quantity " + randomQuantity);

            server.inventory.addRecord(randomID, randomQuantity);
            try {
                this.sleep((int) ((Math.random() * (5000 - 1000)) + 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Test {
    public static void main(String[] args)  {
        ArrayList<TestClientThread> clientThreads = new ArrayList();
        TestServerThread serverThread = new TestServerThread();
        int clientThreadsNum = 50;

        serverThread.start();

        for (int i = 0; i < clientThreadsNum; i++) {
            TestClientThread thread = new TestClientThread();
            clientThreads.add(thread);

            thread.start();
        }
    }
}
```

```

```

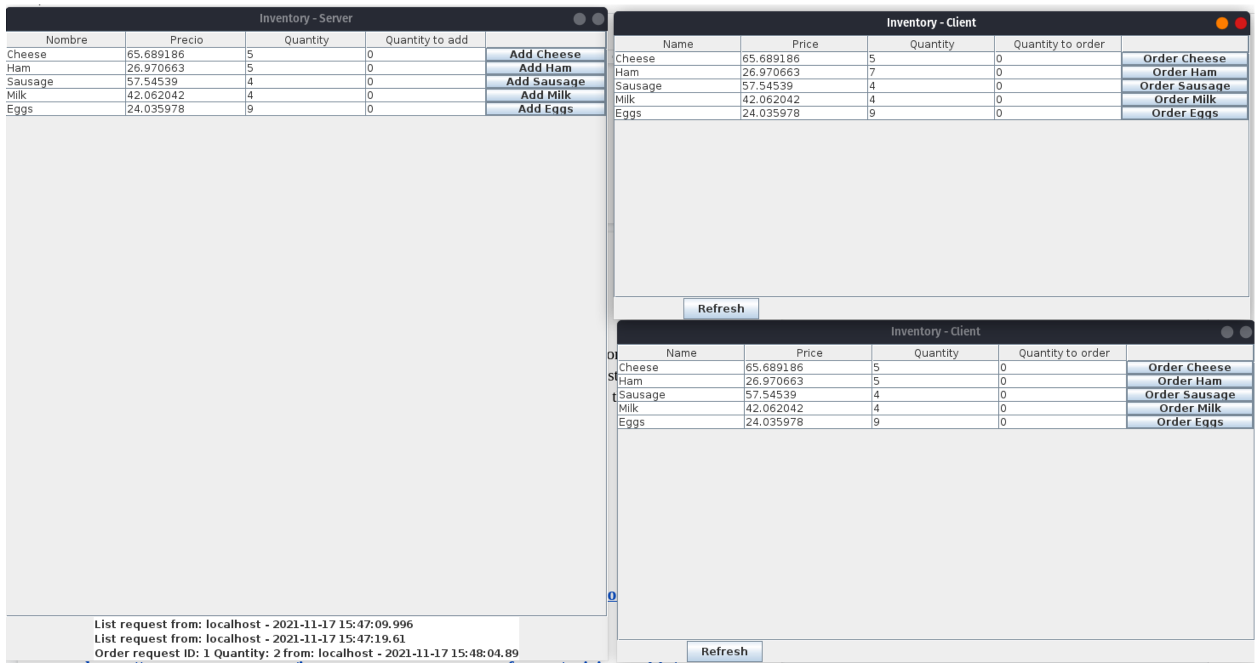
Code Snippet 8. Test subclasses of client and server

The last part to tackle now that every backend part is defined is to create a design for a graphical user interface and implement it with our now abstract classes. Taking in mind that a lot of people use Spreadsheet applications constantly, I have decided to make the graphical interface using primary editable tables so the former users can find it easy to use.

Chapter 3
Results

3.1 Application results

After designing and developing the application, we can now see the application in action. It is important to note that a server instance must be created before a client instance, since the client will attempt to look for the host initially. As established previously, the computer's 6000 port must not be used by another application.



Inventory - Server

Nombre	Precio	Quantity	Quantity to add	
Cheese	65.689186	5	0	Add Cheese
Ham	26.970663	5	0	Add Ham
Sausage	57.54539	4	0	Add Sausage
Milk	42.062042	4	0	Add Milk
Eggs	24.035978	9	0	Add Eggs

```
List request from: localhost - 2021-11-17 15:47:09.996
List request from: localhost - 2021-11-17 15:47:19.61
Order request ID: 1 Quantity: 2 from: localhost - 2021-11-17 15:48:04.89
```

Inventory - Client

Name	Price	Quantity	Quantity to order	
Cheese	65.689186	5	0	Order Cheese
Ham	26.970663	7	0	Order Ham
Sausage	57.54539	4	0	Order Sausage
Milk	42.062042	4	0	Order Milk
Eggs	24.035978	9	0	Order Eggs

Inventory - Client

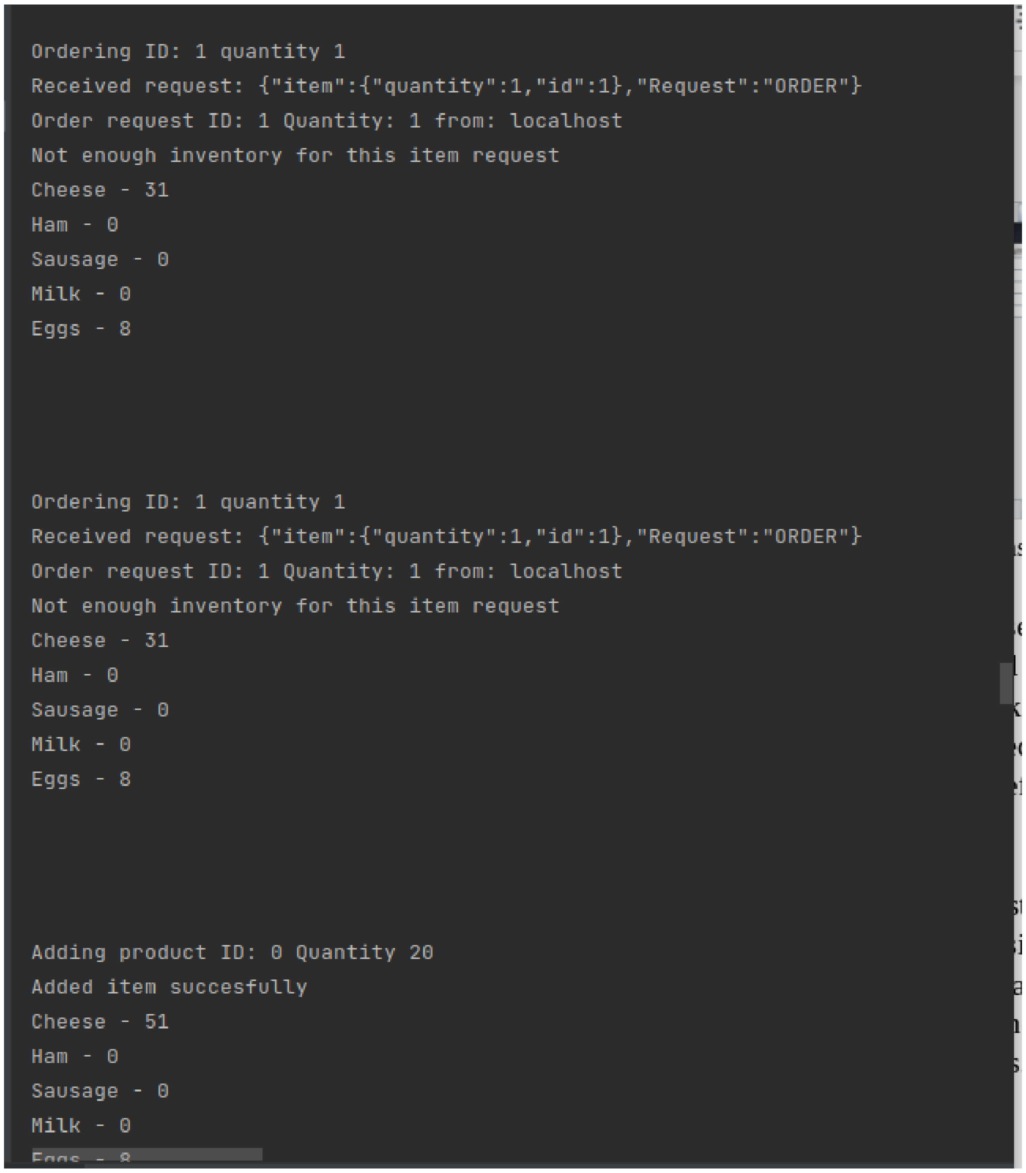
Name	Price	Quantity	Quantity to order	
Cheese	65.689186	5	0	Order Cheese
Ham	26.970663	5	0	Order Ham
Sausage	57.54539	4	0	Order Sausage
Milk	42.062042	4	0	Order Milk
Eggs	24.035978	9	0	Order Eggs

Image 6. Server application (left) and two instances of Client application (right)

As seen on Image 6, both client and server may see a list of items with their current stock. Additionally, a server has an input column to add stock to that specific product while the client can do the opposite if there is enough stock.Noticeable on the bottom of the server application, we are able to see logs from client requests as well in order to have an audit log of transactions. Client instances are able to refresh and fetch potential new data from the server as well.

3.2 Test results

As previously stated, it would be hard to truly test our application with multiple GUIS and assert that the operations are being done simultaneously, which is why we will use our tests developed in the previous section to create 30 client instances in different threads. We will look for any abnormalities or find out if the application stopped abruptly due to our negative stock rule defined in our tests.



```
Ordering ID: 1 quantity 1
Received request: {"item":{"quantity":1,"id":1},"Request":"ORDER"}
Order request ID: 1 Quantity: 1 from: localhost
Not enough inventory for this item request
Cheese - 31
Ham - 0
Sausage - 0
Milk - 0
Eggs - 8




Ordering ID: 1 quantity 1
Received request: {"item":{"quantity":1,"id":1},"Request":"ORDER"}
Order request ID: 1 Quantity: 1 from: localhost
Not enough inventory for this item request
Cheese - 31
Ham - 0
Sausage - 0
Milk - 0
Eggs - 8




Adding product ID: 0 Quantity 20
Added item succesfully
Cheese - 51
Ham - 0
Sausage - 0
Milk - 0
Eggs - 8
```

Image 7. Sample of tests output

While difficult to demonstrate on this written report, we can see from Image 7 a sample of oru tests output. For example, in our second request we can see that a client is attempting to order 1 package of ham but we can see from the previous output that there is no ham available which results in an error message. In the third output, we can see the server simultaneously adding stock to random items in this case adding 20 cheese which previously had 31 stock. Decreasing or increasing the amount of instances didn't affect the test's behaviour. The test class was provided as part of the source code so anyone is free to run it and attempt to spot any outlier.

Chapter 4
Conclusion

4.1 Conclusion

While this inventory management application could be used for some useful features we were able to set up the fundamentals for any potential new type of requests. Using a concurrency paradigm we were able to take advantage of modern computer processor capabilities to handle multiple client requests independently thus reducing any latency that a sequential program would imply for this type of application. Additionally, we were able to test that our application protects our inventory data's integrity from concurrency related problems that can often be one of this paradigm's drawbacks.

4.2 Setup instructions

This project is open source and free for improvement feedback. The source code and executables are hosted on the Github repository:
https://github.com/albertcastaned/InventoryManagment

In order to run the problems, you must have Java Runtime Environment (JRE) installed on your computer. As previously stated on this report, please check that port 6000 is open on your computer and not being used by another application.

A server instance must be initiated before any client. In order to create a server instance, open Server.jar on your computer. Jar files can be executed directly by double-clicking the file, or from the command line depending on your operating system. If your JRE is installed correctly, you should be able to see and control the Server GUI. Once the server has started, multiple instance clients can be opened similarly with the file Client.jar.

Bibliography

Britannica, T. Editors of Encyclopaedia (2019, January 16). Microsoft Excel. Encyclopedia Britannica.
https://www.britannica.com/technology/Microsoft-Excel

History of inventory management technology - city: Clean and simple. CITY. (2020, October 2). Retrieved November 17, 2021, from https://www.citycleanandsimple.com/2018/07/16/history-of-inventory-management-technology/

Inventory management software pricing guide and cost comparison. Inventory Management Software Pricing Guide And Cost Comparison. (n.d.). Retrieved November 18, 2021, from https://www.capterra.com/inventory-management-software/pricing-guide/

MIT. (2014). Reading 17: Concurrency. Retrieved November 18, 2021, from https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/
Concurrency (computer science). Academic Dictionaries and Encyclopedias. (n.d.). Retrieved November 18, 2021, from https://en-academic.com/dic.nsf/enwiki/465314

Javatpoint. (n.d.). Java socket programming (java networking tutorial) - javatpoint. Java Socket Programming. Retrieved November 18, 2021, from https://www.javatpoint.com/socket-programming

Nagarajan, M. (2019, December 2). Concurrency vs. parallelism: A brief view. Concurrency vs. Parallelism — A brief view. Retrieved November 18, 2021, from https://medium.com/@itIsMadhavan/concurrency-vs-parallelism-a-brief-review-b337c8dac350

BillWagner. (2021, September 9). Serialization (C#). Microsoft Docs. Retrieved November 18, 2021, from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/

Introducing json. JSON. (n.d.). Retrieved November 18, 2021, from https://www.json.org/json-en.html

Maven Repository: Org.json " JSON. Maven Repository. (n.d.). Retrieved November 18, 2021, from https://mvnrepository.com/artifact/org.json/json

What is deadlock? - definition from whatis.com. WhatIs.com. (2005, September 21). Retrieved November 18, 2021, from https://whatis.techtarget.com/definition/deadlock

Oracle. (n.d.). Synchronized methods. Synchronized Methods (The Java™ Tutorials > Essential Java Classes > Concurrency). Retrieved November 18, 2021, from https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html.