Inventory System

Joakim Houmøller - 150898 - Jonoe19

Mathias Jensen - 010499 - Maje419

Nicklas Jacobsen - 211098 - Njaco19

We are all accountable for all parts of the report, as it was written in collaboration

November 2021

Contents

Part 1: Approach

Stakeholder analysis

To analyse who the stakeholders are, we drew out the stakeholder salience Venn-diagram, and wrote down different stakeholders on post-its. Then we discussed each stakeholder, and distributed them on the chart.

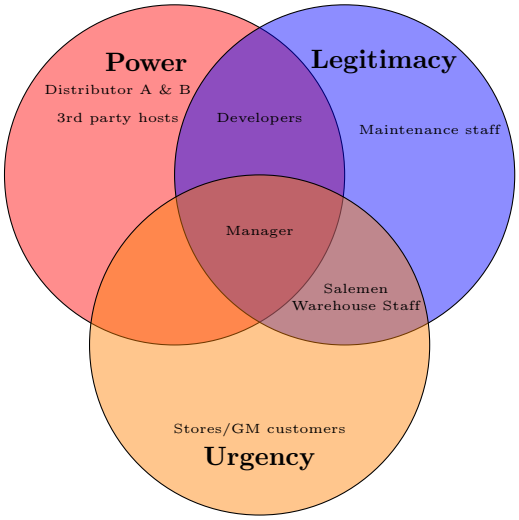


Figure 1: Stakeholder Salience Model of our stakeholders

- **Core stakeholders:**

  GM manager: The manager is highly involved in the process, with a big voice. We assume that this is a guy we need to please, since he can cut funding at any time. He is also our primary source of information, and must therefore be close to the project.

- **Dominant stakeholders:**

  Developers: We have power in the form of leverage, and we have arguably high influence and say in the end product. We are free to make some big decisions about the design of the project, and we likely know when something is possible to implement.

- **Powerful stakeholders:**

  Distributors: Both distributor A and B provide a service to GM, and can deny GM access to the API if they are not satisfied. However, they are not directly involved and have no say in the product.

  Service hosts: The entire product might rely on a server of some sort, which means it will be useless if that service shuts down.

- **Urgent stakeholders:**

  Stores: GM's costumers have no power or influence over the product, but might be incredibly annoyed at the insufficient slow system which is currently in place.

- **Dependant stakeholders:**

  Warehouse staff: The warehouse staff are very dependant on the system, it's a system they have to use on a daily basis.

  Salesmen: Like the warehouse staff, they use this system on a daily basis, and rely heavily on it working. For both of these, it could be argued that the workers also have power, making them core stakeholders. This would depend on whether or not they belong to a workers union. However, this power would mostly affect GM, and not our product. Another consideration is whether it would be smart to have the end-user(s) being a core stakeholder. In this case, we decided that it would not be necessary.

- **Dangerous stakeholders:**

  Politicians: In recent years there have been health scares concerning Teflon cookware, leading to some countries banning the sale of such products. If this happens in GM's market, it could hurt our product.

User analysis

- **End user:**

  Our primary user base will be the ones who need to use the product on a daily basis as an essential part of their job.

  Salesmen: The salesmen use our system to communicate orders from stores to the warehouse staff. They need to use the system every time they do their job. Our system needs to have easy forms to fill, as to ease the workload on the salesmen.

  Warehouse staff: The warehouse staff needs to access the orders that the sales people have put into the system, and need to be able to process these. Our system will improve their life by allowing them easy access to all information they need to do their job.

  Accountant: The accountant will use the system in their day-to-day work to see sales-figures. Having sales and orders in one, simple accessible place will very much ease their work.

- **Super user:**

  The superusers will be any group of users who have more rights than the end user

  Manager: Mr. Manager will likely want access to setting discounts and manage his workers. He will benefit from our system as he will be able to see exactly who completed out what order.

  System administrator (IT department): The IT department is going to need to be superusers, and maybe even have debugging rights, as to work well with the maintainers if a problem arises. They will benefit from knowing about the inner workings of our system, and need to know what they are doing.

- **System user:**

  API systems: At some point, we might want to provide API support for software on different systems.

Who will maintain it?

As one can see in the user analysis, we assume that the day-to-day maintenance of the system can be done by a small IT department at the GM warehouse. Someone on the premises who knows the basic internal structure of the system would be able to solve any issues that might occur. Having knowledge about the computers and especially our system would also allow them to clearly communicate to us any bugs or big problems found with our system.
We, on the other hand, will maintain and expand the code base for the system as long as it is making us money and can be sold to other companies as well. When there are no longer any big or important new features to add, a majority of the development team will move on to other projects, and the system might not be expanded anymore.

Internal and third party systems

Our system might need to provide an API for a mobile app. It will also need to interact with the APIs from distributor A and B. Our system might need to interact with many internal systems as well. GM might have a system for printing labels for their packets, with which we need to integrate. They might also have a system to keep track of finances. We will also need to have our system work with their internal security, be it firewalls or IDS's.

Project Backlog

Now that we've identified our stakeholders and users, it is now time to create, and subsequently structure our backlog. To identify the requirements of the project, we decided to go with user stories. These have the advantage of having the end user in mind, but can sometimes look messy when having too many of them in a backlog. However, we still chose to start the backlog off with them, with the ability to expand the requirements with job, problem, and improvement stories.

We wrote the user stories by splitting up the task between the three developers, where each of us focused on one type of user. The three users we focused on were the end users, namely warehouse workers and salesmen, as well as our core stakeholder, the manager. The focus on these end-users as well as the manager is a intentional strategy, as satisfying the core stakeholder is our prime directive. We also believe that the focus on these 3 user types will help point us towards the development of the MVP, which will likely involve all three of these user types.

Our strategy for writing user stories was simply to write them down on sticky notes, but after having written them here, we discovered the tool "Trello", which provides an interface for structuring our project. We then started using this as our primary project backlog.

When we had our initial user stories, we chose to revisit each one, and discussed how we could split and edit them to be in accordance with the INVEST principles. For instance, we rewrote the user story:

"As a salesman, I want to see the ID of an order, such that I can remove orders with a specific ID"

to:

"As a salesman, I want to be able to distinguish between orders, such that I can manage them separately"

as to follow the N, being negotiable and not defining strictly how to complete the backlog item.

Non-functional requirements:

Non-functional requirements are requirements on how the system operates which are not specific to something the system can do. We found the following non-functional requirements, listed here as user stories:

As a worker I want a UI with easily discernible buttons such that I can learn to use it quickly.

As a user, I want to access the system from my phone, so that i can do work without a PC.

As a warehouse worker, i want to have easy an easy way of filing my work, such that i can do my job quickly.

Structuring the backlog

To structure the backlog, we started out by estimating the effort we believe would be used to complete each backlog item. We defined effort as $effort = complexity + uncertainty + time$. To find the final estimates of effort, we used the technique "Magic estimation" with the Fibonacci numbers as scale. Being completely new to the project, we don't exactly know how many items we could get through in the MVP. Therefore, we decided to start out by picking 12 points worth of backlog items, and put them into a prioritized product backlog. As a team, we agreed on these items, and tried to pick user stories that, when put together, would amount to something like an MVP.

The following items were chosen for the current workplan:

- As a manager I want to have appropriate information about each order such that i am sure everything is correct. (*1 point*)
- As a manager I want to have contact information to each store, such that i can contact them if something goes wrong. (*1 point*)
- As a Warehouse worker I want to pick an order to work on, such that others don't pick the same order. (*1 point*)
- As a salesman I want to see basic store information, such that I can provide better customer service to my clients. (*1 point*)
- As a Warehouse worker I want to complete order such that i can go on to the next order. (*2 points*)
- As a salesman I want to file a new order for an existing client, such that I don't have to type in their information twice. (*3 points*)
- As a Warehouse worker I want to see all orders on a single interface, such that I wont process the same order twice (*3 points*)

Part 2: Development

Designing the MVP

In designing our MVP, we decided to focus on the following line from the interview with GM:

I would know it has become better when we only get orders from the salesmen in one channel

From that line, we understood that having a single channel for warehouse workers and salespeople to communicate would be our MVP. The manager needed to be included as well, as to be able to see that everything had improved.

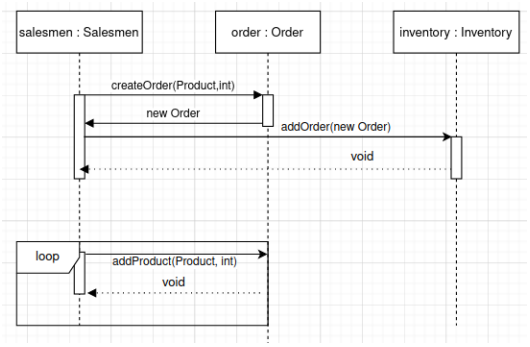
Designing the UML diagrams

To get started with coding, we decided to draw a class diagram documenting a static view of what classes we needed, and how these would interact. We decided to not use noun analysis to find the classes, as all developers had very similar ideas as to how the program should be structured.

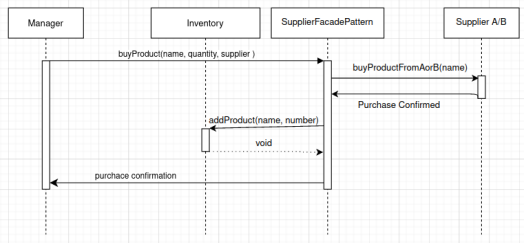
Looking at the current workplan, we identified that Manager, Salesperson and Warehouse Worker would need to be implemented. We also saw the need for implementing a class Store, as well as a class Order to handle storing data about each store and order. The "single channel" of the GM quote will be handled by another class, the singleton "Inventory". This class is a singleton, as to enforce the "single" part of the workplan item.

When working on the class diagram, we also illustrated how we imagine our program will interact with the API of supplier A and B, namely through a class implementing the facade design pattern. This choice was made to simplify dealing with the multiple different formats and API's, and to keep with the SOLID principle of single-responsibility. The facade class will, once implemented, provide a single interface to be called when new items are to be ordered. This class will then figure out how and from which supplier to order them.

The finished class diagram for our MVP can be seen in appendix A (4). Some methods and attributes are prefaced with a minus ('-'), as to indicate that these are not part of the MVP, but to be implemented later.
As for documenting the flows of our program, we used sequence diagrams. Specifically, we documented the dynamic flow of our program when creating an order, as well as when ordering products from supplier A or B:



(a) Creating a new order



(b) Ordering new product

Figure 2: Sequence diagrams depicting the desired flows in our program

For creating an order, we saw from our class diagram that this should be done by a Salesman. We also know that the Inventory class keeps track of all orders. Ordering new products can, for now, only be done by the manager. At some point, it will be doable by the Inventory automatically, but this is not depicted by the sequence diagram, as it is not to be implemented yet. A manager tells the facade pattern what product they want to order, and the facade pattern class takes care of ordering the items from suppliers and adding these to the inventory.

Implementation of the prototype

The MVP, as mentioned above, consists of the classes Order, Inventory, Salesman, Warehouse Worker, Product and Store. This is also what we decided to implement as our prototype. Our inventory system is implemented using IntelliJ, with the programming language Java 13. We chose Java because of the way it handles object oriented programming, as it allows us to create custom objects of a type we define which helps create an easy to understand structure in our code. Java also allows us to implement our prototype following the structure of our UML diagram in an easy manner, since every class in the diagram can be easily implemented as a class in our code. Having already created our class diagram when we began implementing the program allowed us to delegate up the task of programming the classes between the developers, effectively paralleling the effort and speeding up the implementation three-fold. Java also allows us to implement our own exceptions with a custom description for the user, to help them locate the error more easily. This also helps the on-site IT department when handling errors.

We implemented three different users. The manager, the salesman and the warehouse worker each inherits from a common abstract class called Employee.

Testing and coverage

For unit testing we used JUnit, as it provides an intuitive interface for generating test classes. In JUnit, we can implement each test using the **@Test** annotation, and set whether we want to test each class individually or each method. It also generates a coverage report automatically, to make sure every edge case gets tested. We made one test for each method in each class which tested for every case that could occur when using that specific class. In hindsight, we realize that the proper way to do unit testing is to make individual tests for each possible case, so as to identify more efficiently why each test failed. Analyzing the coverage report, we identified untested instructions, and subsequently made further tests to make sure no red code segments remained, except in overriding functions such as *equals()* and *hashCode()* as we didn't deem it necessary. We also skipped running tests on a few lines which only had partial coverage, as we felt that we had met the point of diminishing returns of our coverage, adhering to the mantra of "Run test until uncertainty turns into boredom".

Cyclomatic Complexity

Cyclomatic complexity measures how many paths the code could traverse. To measure it, rather than drawing out a control flow diagram, we use the CodeMR plugin in Intellij to visualize both cyclomatic complexity, but also other important measures such as coupling and cohesion. We then used the CodeMetrics plugin to show the

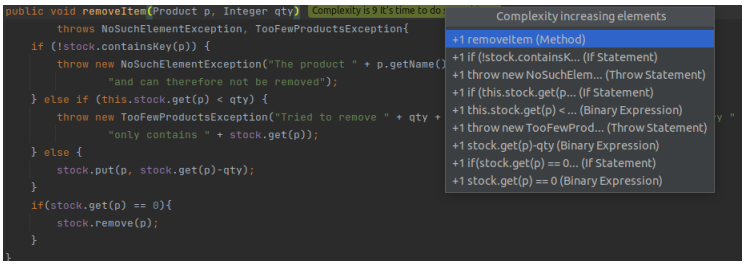


Figure 3: Example of CodeMetric calculations

concrete value of the cyclomatic complexity for each method in our program. CodeMR gave a great overview of the complexity of all our classes. An example can be seen in appendix A 6, but we needed to go through all methods with the CodeMetrics plugin to generate the cyclomatic complexity for these 5. A lot of our methods were hovering around 6-9 complexity, but since this is below 10, and most of the points of complexity were coming from thrown exceptions, we decided that changing these methods would only make the code less understandable. Overall, our functions seemed very simple, and the only function that went above 10 in cyclomatic complexity did so because of a switch statement. For the sake of readability, we decided to not change this method either.

API Specification

The specification of our API for mobile apps is created using Swagger. Swagger allows us to specify the functionality of our API, as well as give us a visual representation of our requests. Our API is able to get a list of all orders with a given order status, which is done via a HTML GET request. It's also possible to get a list of orders with a given salesman and get the details of a specific order given it's ID. Finally it's also possible to add an order to the inventory, which is done via a POST request. All of our specified requests are on level 2 of the Richardson Maturity Scale, as we use several endpoints, and we use the HTTP verbs Get and Post according to good practice.

Architecture

Layering

Our architecture can be described in 2 layers:

- Presentation Layer, our HTML pages
- Business Layer, our program

On the frontend, we used HTML, and only HTML. Any form filled on our front-end, will be sent via a post request to an endpoint to our business layer. Here, the form will be processed, and a new HTML page is returned to the user. At the opposite end, rather than designing and building a database to store the data, we use non-persistent data storage such as ArrayLists and HashMaps, storing everything locally on the server. In a real world setting this would not be ideal, as all inventory and orders would be wiped out if anything turned off the server.

This architecture means we have a cut between the presentation layer and all other layers. The client has no responsibility, and no code is run on the client's machine. This choice was made when we decided to work primarily with Java as our backend.

During the development of the system, our architecture evolved away from the class diagram we drew in the planning phase. This happened dynamically when we found smarter or faster ways of working with the system. As much as this evolution made programming and understanding our system easier, as much did it also make the code rot, since changing the structure of our program risks increasing coupling and lowering cohesion. Fixing bugs and adding new features to the frontend also required restructuring some of our backend flows, and might have also increased the cyclomatic complexity compared to when only the backend functionality was in place.

Distribution patterns

We have used the distributed presentation pattern, as the presentation component is split, meaning two different user types are presented with a different UI. Even though the salesman and the warehouse worker is connected to the exact same backend and can be accessed through the same front-end application, their individual interfaces are distinct. A salesman can only post a new order to the inventory, and the warehouse worker can only process existing orders.

Agile principles

In regards to architecture, the agile principles state clearly one rule for the optimal architecture:

Agile Principle nr. 11

"The best architectures, requirements, and designs emerge from self-organizing teams."

In a traditional organization, there is usually one manager delegating tasks to employees. Similarly, a scrum team has one product owner and one scrum master, meaning the team has overhead delegating tasks to each developer. In our process, there has not been one leader of the project. The process has been organized in such a way that all members owned the product together, where we share a common motivation to deliver. In practice, this means everyone could have their voices heard, leading to a higher quality architecture, while also catering to each member's strengths.

One instance where this power manifested itself was how when we realized that we wanted to change how we store our list of employees. When designing the UML class diagram, we found it intuitive to store the list as an attribute on the manager, as those employees would then be in the manager's control. However, when implementing the software, we realized that accessing the list would be much more intuitive if we stored the list in inventory, so we wouldn't have to fetch a manager object each time we wanted to view the employees. In a traditional- or scrum structure, we would've had to explain the issue to the manager/product owner, and likely be turned down, leading to a lower quality architecture. But since we are a self-organizing team, we quickly unanimously decided to refactor the project, improving the architecture.

UI

Our UI was only HTML in our prototype. Thus, our only UI is what the client browser decides to show. However, all the buttons actually do what they say, and thus, although both our UI and UX are despicable, they are fully functional. We decided to implement a working prototype/simple interface, as this is what we found most exciting. None of the developers have much skill in UX and design, and as such, we decided that the best result would likely come from as working prototype that a specialist could use as groundwork for their design. Orders can only be sorted through the URL, so directly sending a GET request with the status or salesman in the querry. For instance, to find all orders that are made by the salesman "Nicklas", one would type in the url:

localhost:8080/allOrders?salesman=Nicklas

At some point, buttons and filters will be made to do this automatically, but for now, this was too complex and time-consuming to learn to do.

Integrating with suppliers

Since we weren't provided with the two actual API's for the suppliers, we wouldn't get the chance to actually integrate them into the product. However, we will still explain how we would have integrated them in theory.

Since the suppliers use independent API's returning different data formats, the best solution would be to look towards the GoF design patterns. Rather than having a class for each supplier integrating with their specific API, and translating to an agreed upon data format to be used throughout the entire application, we can use the structural facade design pattern.

One class "SupplierFacade" would be responsible for communicating with each supplier, and providing a collection of methods which hide the complexity of communicating with each supplier independently. This way, the manager would only have to specify which supplier to fetch data from, and the facade pattern would take care of the rest. This would result in a higher abstraction, leading to a simpler UX for the manager, but it would also lead to easier implementation of new suppliers in the future.

Since we know that both API's are of Richardson Maturity scale 2, we know that they have implemented the CRUD-operations in http, meaning the facade pattern can access supplier endpoints by sending http requests to their API. The calling sequence for ordering a product from a supplier would work as follows:

First, the facade would authenticate with the supplier according to the RFC7235 HTTP Authentication framework. The facade would send a GET request to the specified supplier, requesting a list of all available products to order. The supplier API would then send status code 200 OK upon success, along with their list of product ID's and available stock. The facade will then submit a POST request for the order GM wants to make, and subsequently the supplier would send a 201 Created upon successful order submission.

Appendix A

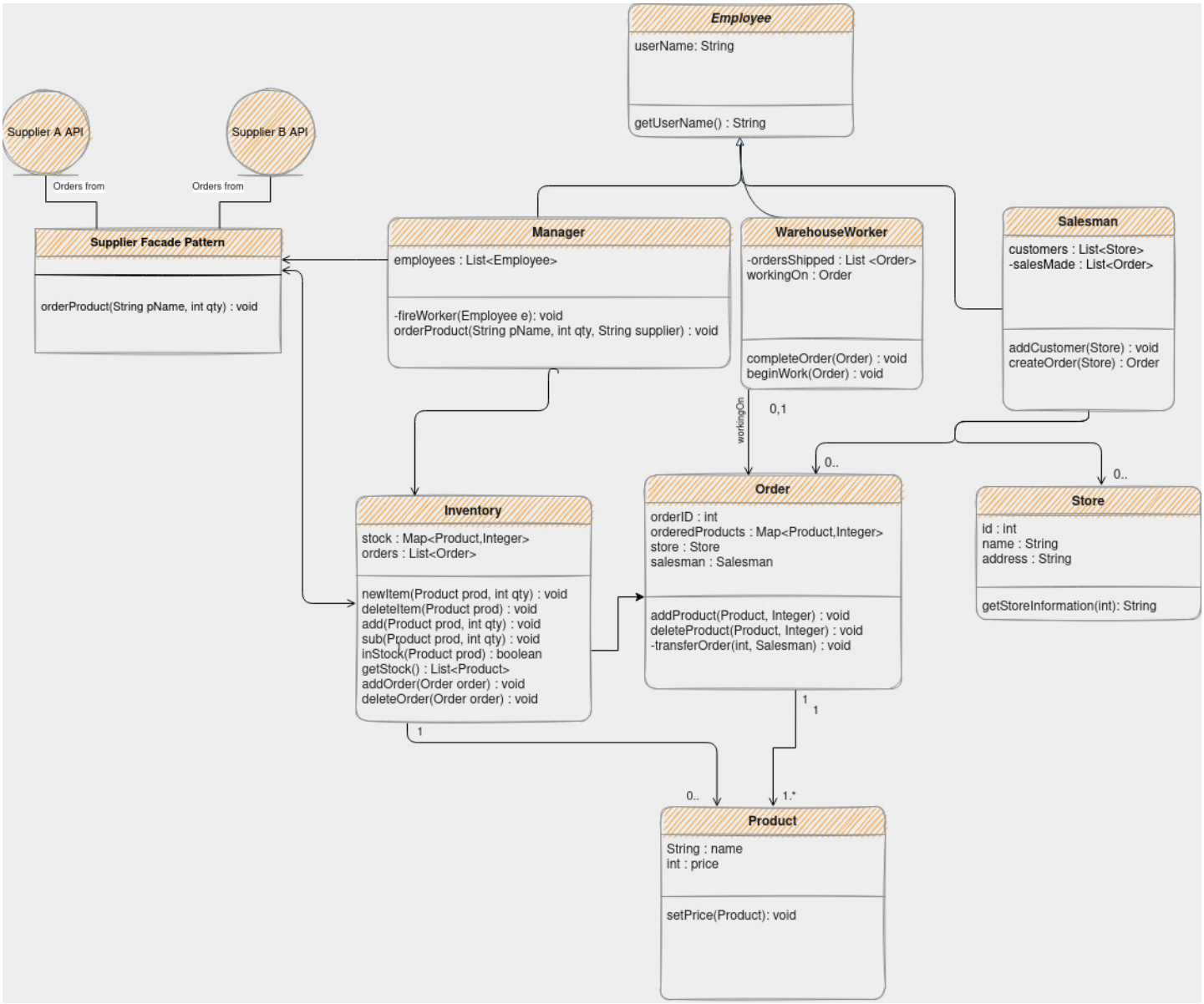


Figure 4: The UML class diagram for our system MVP

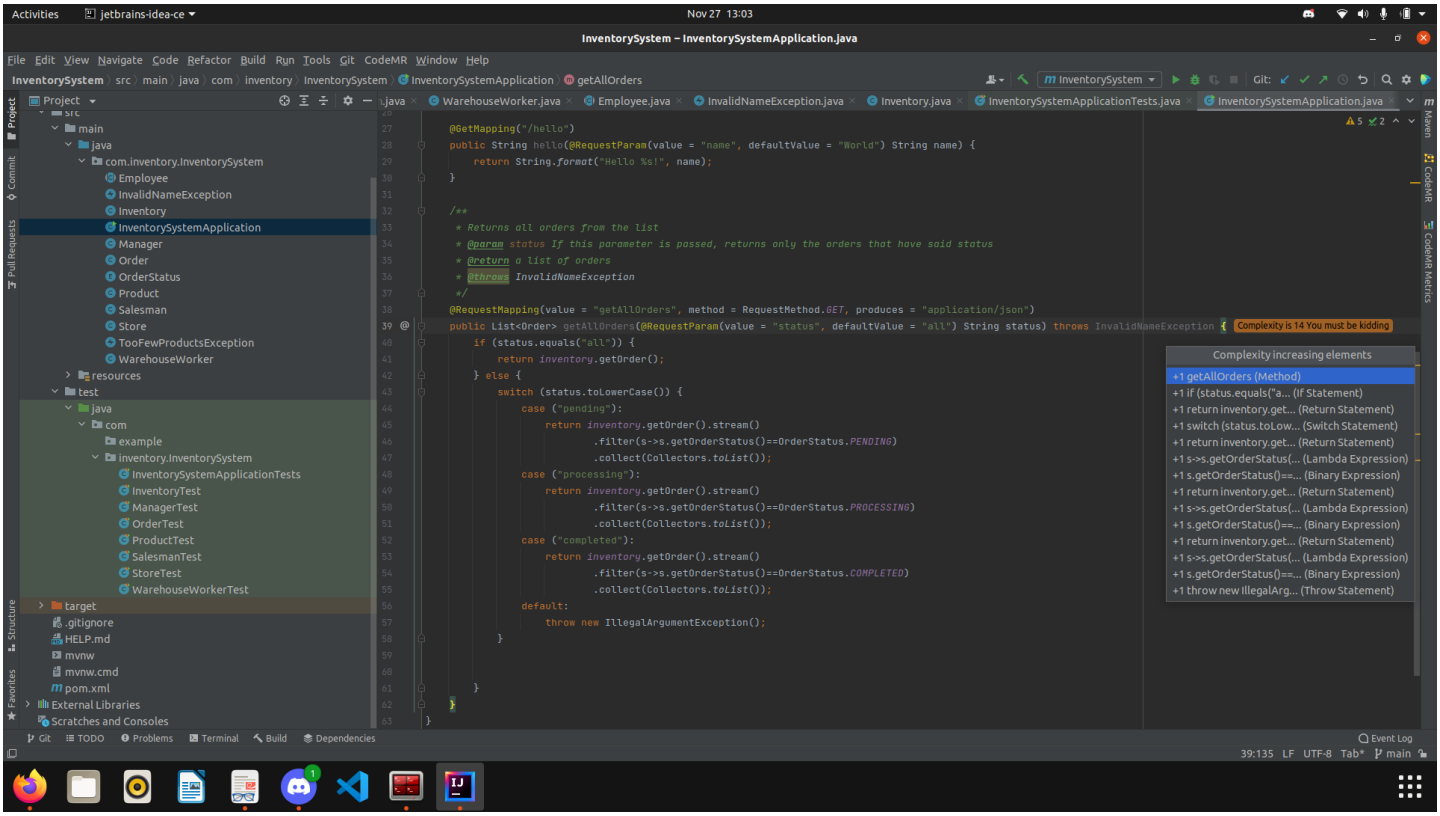


Figure 5: Complexity of our method to send a list of orders to the presentation layer

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	WarehouseWorker					32	low-medium	low	low-medium	low
2	Salesman					31	low-medium	low	low-medium	low
3	Manager					18	low-medium	low	low	low
4	TooFewProductsExc...					6	low-medium	low	low	low
5	InvalidNameException					5	low-medium	low	low	low
6	OrderStatus					4	low-medium	low	low	low
7	Order					69	low	low	high	low-medium
8	Store					53	low	low	low-medium	low-medium
9	Inventory					50	low	low	low-medium	low
10	InventorySystemAp...					35	low	low	low	low
11	Product					33	low	low	low	low
12	Employee					5	low	low	low	low

Figure 6: Overview of our classes from CodeMR