Software Architecture of Optimized Cargo Delivery Tracking System

Aylin İrem Kara-20170703003-072

Meriç Tunahan Tomruk-20190702022

Hasan Barazi - 20170702101

CSE447- Software Architecture

Department of Computer Science Engineering, Yeditepe University

2. Background

Today, increasing trade activities have increased mobility in the logistics industry. The pandemic, which has been experienced around the world, has greatly affected the tendency of consumers to online shopping. This situation has been one of the most important factors that increase the importance of the logistics sector.

One of the most important areas covered by the logistics sector is that the products are delivered to the customer correctly and effectively. For this reason, logistics companies make great investments in product distribution systems. Companies that do not prefer effective and efficient product delivery systems are at a loss. The investment that these companies do not make in order to avoid costs in the future. In addition, these companies are not preferred due to the bad service received by the consumer because these days' consumers want speed, punctuality, undamaged, and trustworthy delivery system.

3. Problem Statement

The product delivery system in the logistics system needs to be improved, costs reduced and made more efficient and effective. These improvements can be addressed with an architectural approach. An architectural approach refers to the overall design and structure of a system or product. In software engineering, an architectural approach involves identifying the components and their relationships, as well as the design patterns that will be used to create a software system. An architectural approach typically involves considering the various constraints and requirements of a project, such as performance, scalability, security, maintainability, and other factors that may impact the design and implementation of the system or product. The chosen approach will often be based on the specific needs and goals of the project, as well as the available resources and budget.

In the world of cargo delivery, finding the shortest path is crucial for maximizing efficiency and minimizing costs. There are various factors that can affect the shortest path for cargo couriers, including distance. One of the most important factors in determining the shortest path is distance. The shorter the distance, the faster and more cost-effective the delivery will be. However, distance is not the only factor to consider. In order to determine the shortest path for cargo couriers, it is important to consider all of these factors and choose the most suitable route based on the needs of the client. By carefully analyzing the options and choosing the most efficient route, cargo couriers can ensure that they are providing the best possible service to their clients.

Overall, cargo delivery systems play a vital role in the global economy, enabling the efficient and timely movement of goods from one location to another. These systems are essential for ensuring that businesses and consumers have access to the products they need. In this article, we focused on an optimized product deployment system considering the software architecture of product delivery systems.

4. Requirements Specification

4.1 Functional Requirements

Shopping From Contracted e-commerce Website: This function is the action of the 'client' shopping from a website that, has an agreement with the cargo delivery firm that uses our system. So, we assume that shopping from these websites will create a database entry automatically. Database entries include some information about the product like its location, destination point, and the buyer, etc.

Call Courier to Home: Sometimes our courier will receive cargo while on the way, customers can give cargo not only from the cargo shop but also from their houses, workplaces... With this function, our cargo guy will take cargo according to route or location.

Give Cargo to Branch: One of the most common and ordinary ways of sending stuff is going to the cargo shop as you know. Also, this system of course includes this service.

Check Cargo Situation: When a cargo guy is on delivery we want to know information like where he is, how many packets he delivered, and how many are left? This function answers these questions synchronically working with the database system and courier which has a GPS on him or on the vehicle.

New Database Entry: This function works only when a new cargo will be taken to the database system. Works with some other functions like shopping from a contracted e-commerce website or giving cargo to a branch (see use case diagram).

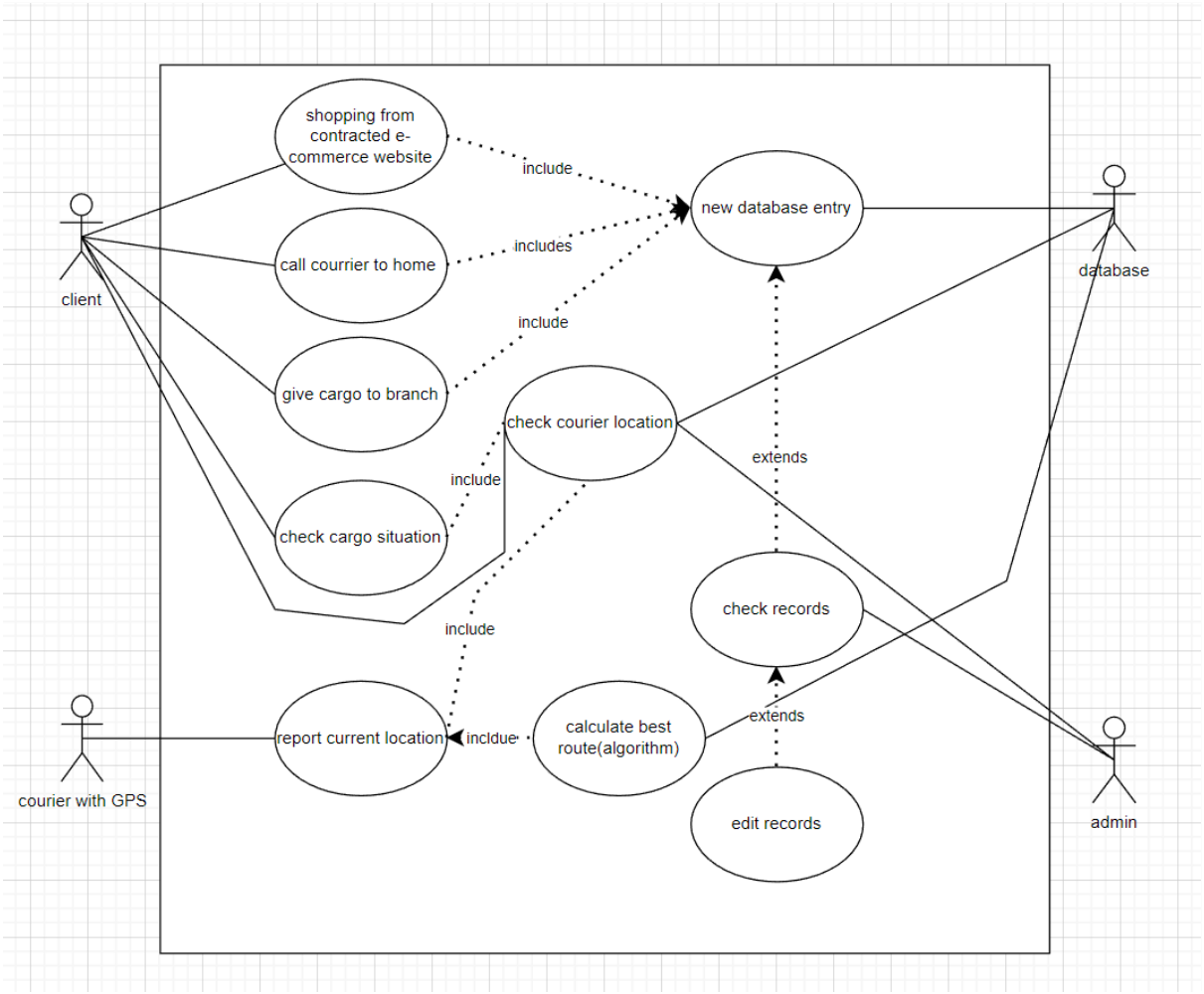
Check Records: In some situations, the admin may want to see the records.

Edit Records: If there is a problem with the records in the database s/he can intervene and solve the problem with the help of this function. It'll update the database due to his moves.

Check Courier Location: This function helps to improve the route that the cargo guy has. If there is a new cargo on the way that enters the system with this function, the algorithm will constantly change the route and improve it. There are also other things that affect the route like traffic situations or path length.

Calculate Best Route(algorithm): This function calculates the best route with the consideration of parameters like the cargo sources, destinations, road, and traffic situation. Calculates how to move between locations so we can serve the fastest and most optimal way possible.

Report Current Location: This function works with a GPS and updates the cargo guy's location on the database so other functions can use this information.



Use Case Diagram

4.2. Non-functional Requirements

Volere Template for Security		
Requirement No.:1	**Requirement Type:**	**Use Cases:**
Description: The product shall keep user information private.		
Rationale: Users' personal information can be captured in any security vulnerability. For this reason, security must be ensured.		
Source:		
Fit Criterion:		
Customer Satisfaction Rating:	**Customer Dissatisfaction Rating:**	
Dependencies:	**Conflicts:**	
History:	**Supporting Materials:**	

Volere Template for Usability		
Requirement No.:2	**Requirement Type:**	**Use Cases:**
Description: The system requires it to be easy to use.		
Rationale: All stakeholders can be able to use the system.		
Source:		
Fit Criterion:		
Customer Satisfaction Rating:	**Customer Dissatisfaction Rating:**	
Dependencies:	**Conflicts:**	
History:	**Supporting Materials:**	

Volere Template for Efficiency		
Requirement No.:3	**Requirement Type:**	**Use Cases:**
Description: It has to be working efficiently		
Rationale: The system ensures the best performance under undesirable conditions and should prevent wasting materials, energy, efforts, and money. The necessary inputs should be taken and the response should be given within the minimum delay.		
Source:		
Fit Criterion:		
Customer Satisfaction Rating:	**Customer Dissatisfaction Rating:**	
Dependencies:	**Conflicts:**	
History:	**Supporting Materials:**	

Volere Template Maintainability		
Requirement No.:4	**Requirement Type:**	**Use Cases:**
Description: The system should be maintainable and capable of being maintained easily.		
Rationale: Errors that may occur in the system should be detected and solutions should be applied. The system should be easily maintainable throughout its lifetime and usable after maintenance.		
Source:		
Fit Criterion:		
Customer Satisfaction Rating:	**Customer Dissatisfaction Rating:**	
Dependencies:	**Conflicts:**	
History:	**Supporting Materials:**	

Volere Template for Reusability		
Requirement No.:5	**Requirement Type:**	**Use Cases:**
Description: The product has to be reusable.		
Rationale: Its parts can be needed in other projects so less programming(resource) is needed.		
Source:		
Fit Criterion:		
Customer Satisfaction Rating:	**Customer Dissatisfaction Rating:**	
Dependencies:	**Conflicts:**	
History:	**Supporting Materials:**	

Volere Template for Scalability		
Requirement No.:6	**Requirement Type:**	**Use Cases:**
Description: The system should be scalable and flexible.		
Rationale: The system performs when the number of users increases or decreases, and how well the database handle capacity.		
Source:		
Fit Criterion:		
Customer Satisfaction Rating:	**Customer Dissatisfaction Rating:**	
Dependencies:	**Conflicts:**	
History:	**Supporting Materials:**	

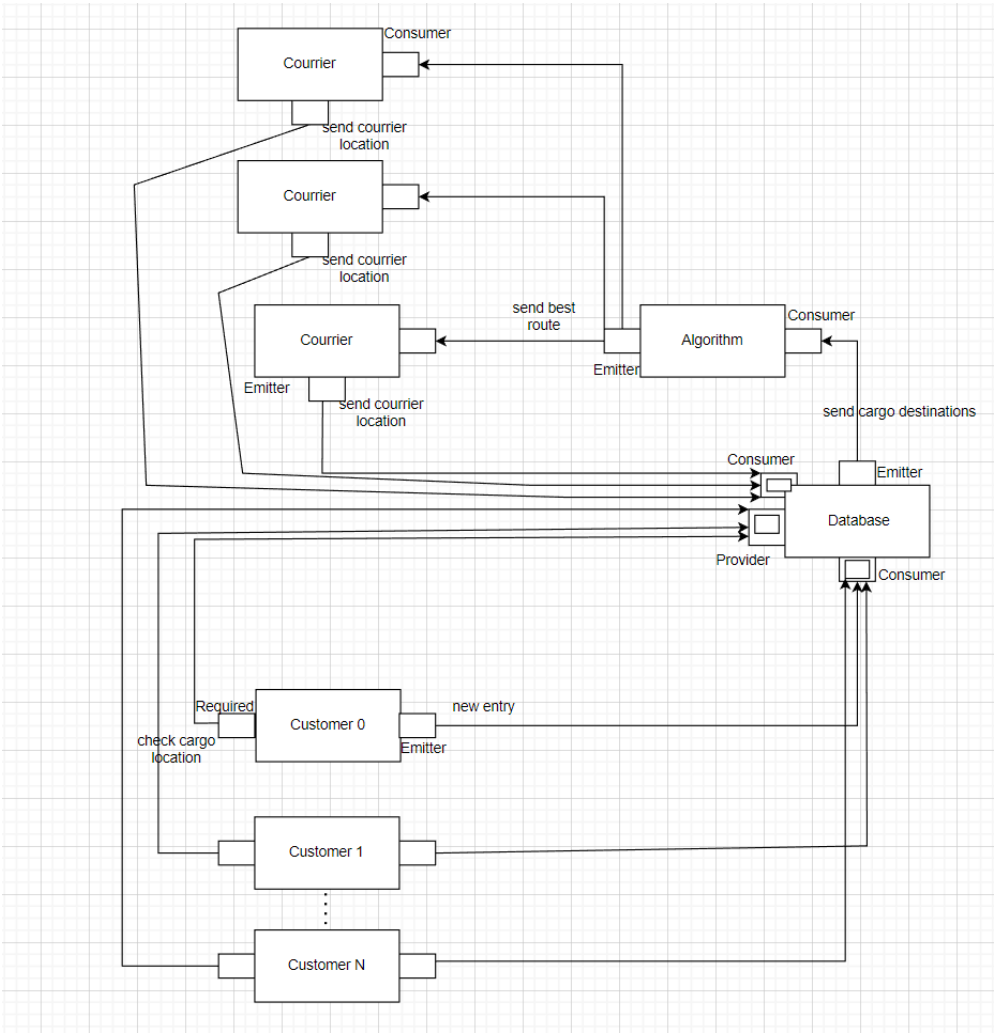
Volere Template for Performance		
Requirement No.:8	**Requirement Type:**	**Use Cases:**
Description: The system should be a high-performance one.		
Rationale: To have a short response time, high throughput, and be fast.		
Source:		
Fit Criterion:		
Customer Satisfaction Rating:	**Customer Dissatisfaction Rating:**	
Dependencies:	**Conflicts:**	
History:	**Supporting Materials:**	

Volere Template for Modularity		
Requirement No.:6	**Requirement Type:**	**Use Cases:**
Description: The system should be considered modular.		
Rationale: The system can be easily disassembled into smaller parts and must be able to be associated with other relevant parts.		
Source:		
Fit Criterion:		
Customer Satisfaction Rating:	**Customer Dissatisfaction Rating:**	
Dependencies:	**Conflicts:**	
History:	**Supporting Materials:**	

5. Architecture Specification

5.1. Structure

The cargo delivery tracking system is considered in this study. This system consists of a number of N customers and courier components. In the system, there are algorithm and database components.



Conceptual Diagram

Database Component

The database component has four ports. One of these ports is the consumer port array of size N which receives new entries of cargo requests from consumer ports. Another port is an emitter to send cargo destinations to the algorithm for the inputs of calculating the best routes. The third port of the database component is a consumer port that receives couriers' current locations from the courier components. Also, the database component has provided a port array of size N to receive cargo location checkings from customers.

Customer Components

The system has N customer components with two ports. One of these ports is the emitter port to create new entries to the database for recording cargo orders' destinations. Another port is the required port that checks cargo locations from the database.

Algorithm Component

There are also algorithm components with two ports. One of them is the emitter port to send the best routes, calculated by the algorithm, to follow by couriers. The second port is a consumer port that receives cargo destinations from the system's database.

Courier Components

The system has a number of N courier components. Each of these has consumer and emitter ports. The consumer ports take the best routes calculated and sent by the algorithm. Also, the emitter ports send couriers' current locations to the database.

5.2. Behaviour

5.2.1. XCD Architecture Specification for the Cargo Delivery Tracking System

5.2.1.1. Components Specifications

Courier Components Specification

```
component courier (int courierID N:=10){
        String GPSdata = NULL;
        String courierLocation[@] = GPSdata;
        bool sendBestRoute = false;
        bool DataTaken := false;
        bool courierAssigned = false;
    consumer port toAlgorithm{
            @interaction{
                waits:sendBestRoute != true;
            }
            @functional {
                requires:DataTaken == false;
                ensures: \result :courierLocation[@];
                                sendBestRoute = true;
            }
            takeBestRoute();
        }
        emitter port toDatabase{
            @interaction{
                waits:courierAssigned!=true;
```

```
        }
        @functional{
            requires:courierID \in [0,N-1];
                GPSdata := courierLocation[courierID];
            ensures:\result: GPSdata;
        }
        sendCourierLocation();
    }
}
```

Algorithm Component Specification

```
component Algorithm{
int data = 0;
bool calculateRoute:=false;
emitter port toCourier(ID:=N){
    String Route[]:=null;
@interaction{
    waits: calculateRoute == true;
}
@functional{
    requires: haveLocations==true,TrafficSituation(),CourierLocation();
    promises: Route[] = Algorithm(data);
    ensures: Route[]:=!null;
```

```
}
calculateRoute();
sendBestRoute();
sendRouteInfo();
}
consumer port fromDatabase(int orderID N:= 50){
        String Locations[N]:=null;
@interaction{
        waits: Locations[@]:=data;
}
@functional{
        promises: Locations[@] = CargoDestinations;
        ensures: Locations[@]:=!null;
}
takeCourierLocation();
takeCargoDestinations();
sendTrafficSituation();
}
}
```

Customer Components Specification

```
component customer(){
   bool orderMade := false;
   bool locationQueryMade := false;
   String customerAddress := None;
```

```
    emitter port toDatabase{
        @interaction{
            waits: !orderMade;
        }
        @functional {
            promises: address = customerAddress; //input taken from customer
            ensures: orderMade := true;
        }
        newEntry(orderID);
    }
    required port fromDataBase{
        @interaction{
            waits: locationQueryMade != true;
        }
        @functional{
            requires:\result == orderID;
            ensures: locationQueryMade := false;
        }
        checkCargoLocation(orderID);
    }

}
```

Database Component Specification

```
component database(int orderID N:= 50, int courierID N:= 10){
        String courierLocation[N] := None;
    registered[N] := false;
    bool queryMade[N] := false;
    String destination[N] := None;

    emitter port toAlgorithm{
        @interaction{
            accepts: destination[@] != None;
        }
        @functional{
            requires: true;
            ensures: \result: destination[@];
        }
        sendCargoDestinations();
    }
    consumer port fromCourier[int courierID] {
        @interaction{
            accepts: courierLocation[@] == None;
        }
        @functional{
            requires: true;
            ensures: courierLocation[@] := address_arg;
```

```
        }
        receiveCargoLocations();
    }
    provided port fromCustomer[int orderID]{
        @interaction{
            accepts: queryMade[@] == true;
        }
        @functional{
            requires: courierLocation[@] != None;
            ensures: queryMade[@] := false;
                    \result := courierLocation[@];
        }
        checkCargoLocation(orderID);
    }
    consumer port fromCustomer[int orderID]{
        @interaction{
            waits: registered[@] == false;
        }
        @functional{
            requires: true;
            ensures: registered[@] := true;
        }
        newEntry();
    }
}
```

5.2.1.2. Connector Specification

```
connector courier2database(Courier{toDatabase}, Database{fromCourier}){
        role Customer{emitter port toDatabase{CourierLocation();}}
        role Database{consumer port fromCourier{CourierLocation();}}
        connector courier2db_location(Courier{toDatabase}, Database{fromCourier});
}

connector customer2Database1(Customer{toDatabase}, Database{fromCustomer}){
   role Customer{emitter port toDataBase{newEntry();}}
   role Database{consumer port fromCustomer{newEntry();}}
   connector cust2db_newEntry(Customer{toDatabase}, Databse{fromCustomer});
   }
}

connector customer2Database2(Customer{fromDatabase}, Database{toCustomer}){
   role Customer{required port toDataBase{checkCargoLocation(int courierID);}}
   role Database{emitter port fromCustomer{checkCargoLocation(int courierID);}}
   connector cust2db_checkCargoLocation(Customer{fromDatabase},
Database{toCustomer});
   }
}

connector Algorithm2Courier(Algorithm{toCourier},Courier{fromAlgorithm})
```

```
{
        role Algorithm{ emitter port toCourier{sendRouteInfo();} }
        role Courier{ consumer port fromAlgorithm{takeRouteInfo();} }

        connector algorithm2courier (Algorithm{toCourier},Courier{fromAlgorithm});
}
connector Database2Algorithm(Database{toAlgorithm},Algorithm{fromDatabase}){
        role Database{emitter port toAlgorithm{sendCargoDestinations();}}
        role Algorithm{consumer port fromDatabase{takeRouteInfo();}}
        connector database2algorithm (Algorithm{fromDatabase},Database{toAlgorithm});
}
```

5.2.1.3. Software Configuration Specification

```
component CargoDeliveryTrackingSystem(orderID N := 50, courierID M := 10 ) {
        component customer customerIns[N] ( );
        component courier courierIns[M] ( );
        component algorithm algorithmIns ( );
        component database databaseIns (N,M);
}
```

6. Traceability between Requirements and Architecture

To connect our requirements and design decisions, we need to consider the goals and needs of the system we designed and how the design decisions we make will support the requirements of the system.

Functional requirements are specific features or capabilities that a system must have in order to meet the needs of the user or stakeholder. For example, a functional requirement for a cargo delivery and tracking system might be having the ability to track the location of packages in real time.

That is why in our system, the courier and its GPS-integrated vehicle constantly update its data, such as; current location, delivered or taken cargo, to the database, and because the component ‘customer’ is already connected to the component database, it can become possible for customers to see the information about their packages, etc. We were carefully designing this system so there wouldn’t be any deadlocks or other problems. Although there were many things that were related to each other, hence needed to be connected to use their data, we followed the principle “Separation of Concerns” so the system can be more precise about the way it works and both can be understood by others more easily and the system will be more maintainable and also less work will be done in the end.

There are 3 functional requirements we have that are very simple and similar to each other in terms of how they work. These are ‘shopping from contracted e-commerce website’, ‘call the courier to home’, and ‘give cargo to branch’. These 3 in the use just for taking cargo. In the end, they have similar properties (mainly they will create a new entry that has an address). So they work with the component database and courier. But they have nothing to do with the Customer because the database will handle the work. In our system customer has no job with the courier directly. But they communicate via a database.

The database is more of a passive component actually. It doesn't make any decisions or something but may be used by an admin to check the situation in the database and may even change them. Usually, the algorithm component uses a database to calculate the route and sends it to the courier. But there are also other uses of it like the customer checking his/her cargo location.

When it comes to the non-functional requirements of the system, we already determined them as; security, usability, efficiency, maintainability, reusability, modularity, performance, and scalability.

The most important non-functional requirement is making a secure system. There are a few things we would do. First, the data in the database must be encrypted because they are so important. This information about users includes credit cards, addresses, names, etc. A rigorous signing-up process including authentication would help too. These things are the part of database component and can be provided easily with the services of companies like IBM (authentication, encryption…).

Usability, it's about making a usable, aesthetic, easy system. The interface must be plain looking and should only serve its purpose. This doesn't directly affect this project because we only worked on a system and how it works. But in a real-life project UI is an essential aspect and should be handled carefully.

Our system is also efficient and maintainable. Although it is a very simple plain system because there are minimum coupling as possible, every component is by itself. For example, for the functions like "call courier to home" and "check cargo situation" instead of connecting courier and customer to each other, they use the database as a middleman. Since these components are already in cooperation with each other, it's way easier to handle all the jobs this way instead of finding new ways to work with other components work in harmony. So, it

is a clean system and this also benefits when it comes to improving the project with new functionalities. Is also valid for performance since there are not many complex calculations or processes.

Since we already separated every component and the functionalities of our system. And as well as the functions have not adhered to anything. It is easy and possible to use part of this system in other projects.

Sometimes a system can't meet the user demands. That is why we need scalability. For this system horizontal scaling is appropriate. So if there is a problem with a machine the others can take the load.

7. Model-to-Code Transformation Algorithm Specifications

```
function trackCargoDelivery(courierID, orderID):
    # Initialize variables
    distance = 0
    path = [ ]
    unvisitedNodes = set of all nodes in the delivery network
    currentNode = starting node
    previousNode = null

    # Set the distance of the starting node to 0
    distances[currentNode] = 0

    # While there are unvisited nodes
    while unvisitedNodes is not empty:
        # Find the unvisited node with the smallest distance from the starting node
```

```
    currentNode = smallestDistanceNode(unvisitedNodes, distances)

    # If the current node is the destination node, we have found the shortest path
    if currentNode == destinationNode:
        # Construct the path by following the previous nodes
        path = constructPath(previousNode, currentNode)
        return path, distance

    # Remove the current node from the unvisited nodes
    unvisitedNodes.remove (currentNode)

    # Update the distance to all neighboring nodes
    for neighbor in neighbors(currentNode):
        # Calculate the distance to the neighbor through the current node
        distance = distances[currentNode] + distance(currentNode, neighbor)
        # If the calculated distance is less than the current distance to the neighbor, update it
        if distance < distances[neighbor]:
            distances[neighbor] = distance
            previousNode[neighbor] = currentNode

# If we reach this point, it means there is no path to the destination
return "No path found"
```

This pseudocode assumes that you have implemented the following helper functions:

smallestDistanceNode (unvisitedNodes, distances): returns the unvisited node with the smallest distance from the starting node

constructPath (previousNode, currentNode): returns the path from the starting node to the current node by following the previous nodes

neighbors (currentNode): returns a list of the neighbouring nodes of the current node

distance (currentNode, neighbor): returns the distance between the current

8. Lessons Learned

- Software projects should not start with direct coding. At first, the requirements should be clearly defined and a software architectural approach should be applied.
- Implementing software architecture ensures that the errors and risks that may arise in the system can be predicted in advance. Thus, measures can be taken against them and requirements can be defined.
- It is important to determine the units which are the components and the connections between them when starting the software processes.
- Using tools such as use case and natural language definition prepared in this software architecture project ensures that software projects are understood by everyone. This strengthens the communication of team teams with different knowledge levels working together.
- As in this project, because the software architecture is component-based helps to define the interfaces between different components of the system. When associating related units, it becomes easier to express the relationship of different units.

- Using “Architecture Design Language” like XCD. It helped us the describing our system architecture. Since it is formal and precise ADL’s give no room for misunderstanding or misinterpretation.
- How a system works thoroughly, what are the relations between the system components, and how it affects other things.
- That it is a team job and a project should be considered by many others and should have a brainstorm to see different perspectives to a system.
- Architecture design decisions can be and should be checked with the model checkers so there wouldn’t be problems.

9. Conclusion

In this study, the cargo delivery tracking system is considered. This system was optimized and this optimized system was handled with software architecture. As a result of the studies, besides reducing the cost of the system, the benefits of using software architecture were encountered. The architectural approach enabled the improvement and quality of the decisions to be taken in this system. Non-functional and functional requirements were determined and implemented. Since it is important to examine these requirements, they are explained with the chosen techniques. Expressing functional requirements visually by preparing a use case diagram made it understandable for everyone. The expression of non-functional requirements provided clarification of what is expected of the system in many ways. The system was first expressed using natural language. In this way, although the system has become understandable by everyone, it was expressed with XCD because it could facilitate different understandings because clear syntax and semantics were not used. Since

this software architecture modelling language has a clear syntax and semantics, its system and requirements were clearly stated.

Today, there is a need for quality systems in the logistics industry, which has suffered a lot due to the lack of various systems. In this study, the cargo tracking delivery system, which is one of the fields of interest in the logistics industry, is optimized and software architecture is expressed in accordance with its requirements. Instead of coding the system first, it is very important to implement the software architecture and determine the requirements correctly, as was done in this study. Thus, it is aimed to establish a quality, and construct a modular, reusable, and maintainable system.