KOCAELİ ÜNİVERSİTESİ BİLGİSAYAR MÜHENDİSLİĞİ

PROGRAMLAMA LABORATUVARI - II
PROJE – I

ULAŞIM ROTA PLANLAMA SİSTEMİ

Saffet Hakan KOÇAK - 230202058 Yusuf BÜLBÜL - 230202050

I. ÖZET

Proje kapsamında, şehir içi toplu taşıma ve taksi hizmetini birleştiren dinamik bir rota planlama sistemi geliştirilmiştir. Sistem, kullanıcıların interaktif harita üzerinden başlangıç ve varış noktalarını belirlemesi, yolcu tipi ve ödeme yöntemi gibi parametrelerin rota optimizasyonunda rol oynaması üzerine kuruludur. Dijkstra algoritması; mesafe, süre veya ücret ağırlıklı en uygun rotayı hesaplamak için kullanılmıştır. Proje Java programlama dili kullanılarak Visual Studio Code idesinde yazılmıştır.

II. GİRİŞ

Günümüz şehirlerinde artan nüfus ve hızlı kentleşme, ulaşım sistemlerinde esnek, verimli ve kullanıcı odaklı çözümler geliştirmeyi zorunlu kılmaktadır. Geleneksel modeller, yoğunluk, zaman ve maliyet faktörlerini yeterince karşılayamadığından, farklı taşıma modlarını entegre eden dinamik rota planlama sistemlerine ihtiyaç duyulmaktadır.

Bu projede, kullanıcıların interaktif harita üzerinden başlangıç ve varış noktalarını belirlediği; yolcu tipleri (Öğrenci, Yaşlı, Genel) ve ödeme yöntemleri (Nakit, Kredi Kartı, Kentkart) gibi parametrelerle en uygun güzergahı seçebildiği bir sistem geliştirilmiştir. Dijkstra algoritması ve strateji deseni ile güzergah, ücret, süre veya mesafe gibi kriterlere göre optimize edilmiş, Spring Boot tabanlı RESTful API'ler ve Leaflet destekli arayüz sayesinde kullanıcılar gerçek zamanlı rota detaylarına ulaşabilmiştir.

Bu rapor, projenin tasarım aşamaları, kullanılan yöntemler ve sistem entegrasyonu süreçlerini özetleyerek, modern kentlerin ulaşım sorunlarına kullanıcı odaklı çözümler sunduğunu göstermektedir.

III.YÖNTEM

Bu projede, şehir içi ulaşımda kullanıcıya en uygun rotayı sunmayı amaçlayan dinamik ve modüler bir sistem geliştirilmiştir. Projede kullanılan yöntem ve teknikler aşağıdaki ana başlıklar altında detaylandırılmıştır:

III.A) Veri Modeli ve Temel Yapılar

Nesne Yönelimli Yaklaşım: Projede temel kavramlar (örneğin, Araç, Yolcu, Ödeme Yöntemi) soyut sınıflar ve arayüzler aracılığıyla modellenmiştir. Böylece farklı türlerdeki araçlar (otobüs, tramvay, taksi), yolcu tipleri (öğrenci, yaşlı, genel) ve ödeme yöntemleri (nakit, kredi kartı, kentkart) için esnek bir yapı oluşturulmuştur.

Veri Modeli: Şehir verileri, durak bilgileri, bağlantılar (edge) ve transfer detayları gibi bilgiler, CityData, Stop, EdgeInfo, RouteEdge, NextStopInfo ve Transfer gibi sınıflarla temsil edilmiştir. Bu yapı, verilerin okunması ve işlenmesi için sağlam bir temel sağlar.

III.B) Graf Tabanlı Rota Hesaplama

Graf Oluşturma: GraphBuilderService sınıfı, JSON formatındaki şehir verilerini Gson kütüphanesi kullanarak okur ve verileri, her biri bir durak (vertex) ve duraklar arasındaki bağlantılar (edge) şeklinde temsil edilen ManualGraph yapısına dönüştürür. Bu graf yapısı, rota hesaplamalarında kullanılır.

Dijkstra Algoritması: DijkstraSolver sınıfı, graf üzerinde en kısa yolu bulmak için klasik Dijkstra algoritmasını uygular. Ağırlık olarak mesafe, süre veya ücret gibi parametreler kullanılabilmekte, böylece kullanıcı tercihine göre farklı rota hesaplamaları yapılabilmektedir.

III.C) Strateji Tasarımı ve Uygulaması

Strateji Deseni: Rota hesaplama işlemleri için RouteStrategy arayüzü tanımlanmış ve bu arayüz farklı rota stratejileri (örneğin, CheapestRouteStrategy, FastestRouteStrategy, ShortestRouteStrategy, BusRouteStrategy, TramRouteStrategy, TaxiRouteStrategy) tarafından uygulanmıştır. Bu desen, sistemin farklı senaryolara kolayca adapte olabilmesini sağlar.

Dinamik Ağırlık Seçimi: Her strateji, ilgili ağırlık parametresine göre (ücret, süre, mesafe) graf üzerinde hesaplamalar yapar. Böylece kullanıcı seçimine göre esnek ve dinamik rota sonuçları elde edilir.

III.D) Kullanıcı Seçimleri ve Entegrasyon

Kullanıcı Parametreleri: Kullanıcı, web arayüzü üzerinden yolcu tipi, ödeme yöntemi, başlangıç ve varış noktalarını seçer. Bu seçimler, REST API uç noktaları (örneğin, selectPassengerType, selectPaymentType, setStartPoint, setDestinationPoint) aracılığıyla arka uç sistemine iletilir.

İndirim ve Ek Ücret Hesaplamaları: Seçilen yolcu tipine göre (öğrenci, yaşlı) indirimler; ödeme yöntemi olarak seçilen kredi kartı veya kentkartı ise ek ücret (zam) veya indirim uygulamalarını tetikler. Bu hesaplamalar, RotaHesaplama sınıfı içerisinde dinamik olarak gerçekleştirilir.

III.E) Ön Yüz ve Harita Entegrasyonu

Leaflet ile Harita Görselleştirmesi: HTML, CSS ve JavaScript kullanılarak oluşturulan arayüzde, Leaflet kütüphanesi sayesinde harita üzerinde duraklar, rotalar ve kullanıcı tarafından seçilen noktalar görselleştirilir.

Rota Çizimi ve Bilgi Sunumu: Hesaplanan rota, HTML formatında dinamik olarak oluşturulup kullanıcıya sunulur. Bu rota bilgisi, harita üzerinde polyline'lar ve ikonlarla detaylandırılarak görsel bir anlatım sağlanır. Örneğin, otobüs, tramvay ve taksi rotaları farklı renk ve desenlerle ayrıştırılır.

III.F) Uygulama ve Test Süreci

Modüler Geliştirme: Her bir modül (veri okuma, graf oluşturma, rota hesaplama, kullanıcı entegrasyonu) bağımsız olarak geliştirilmiş ve birbiriyle uyumlu çalışması sağlanmıştır. Bu sayede projenin ölçeklenebilirliği ve bakımı kolaylaşmıştır.

Entegrasyon Testleri: REST API uç noktalarının ve harita entegrasyonunun doğru çalıştığı, kullanıcı seçimlerine göre doğru rota hesaplamalarının yapıldığı çeşitli test senaryoları ile doğrulanmıştır.

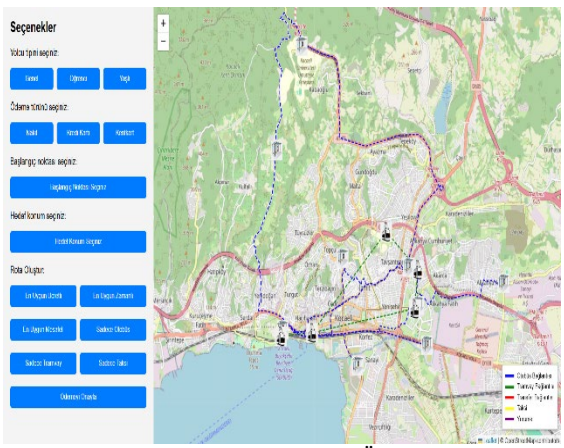

Fig.1 Kullanıcı Seçimi Öncesi

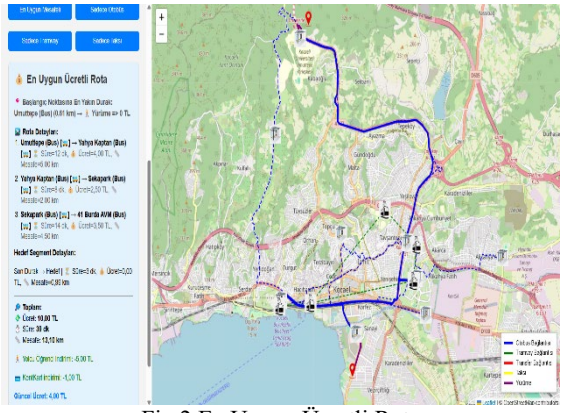

Fig.2 En Uygun Ücretli Rota

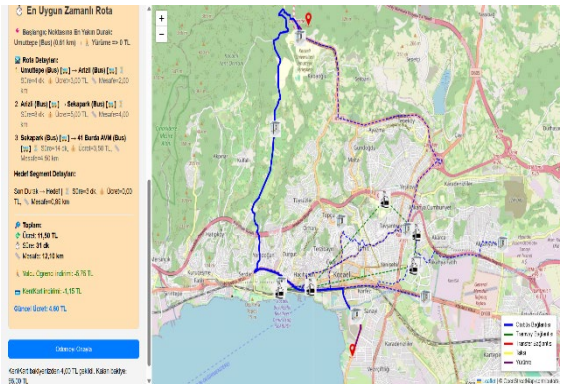

Fig.3 En Uygun Zamanlı Rota

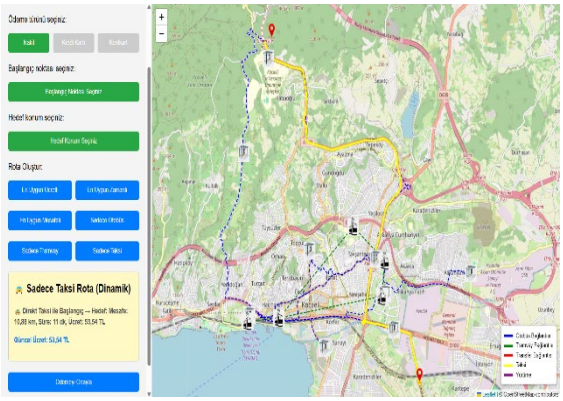

Fig.4 Sadece Taksi Kullanımı

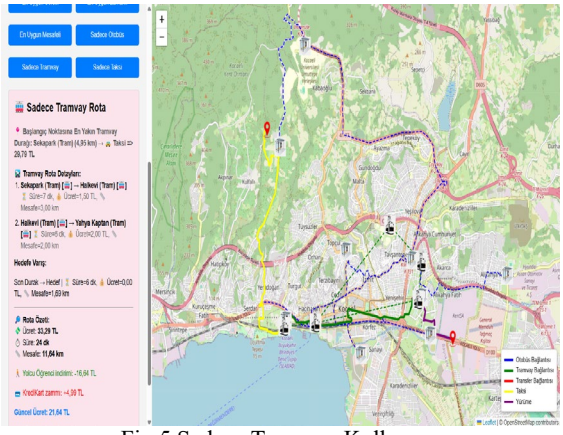

Fig.5 Sadece Tramvay Kullanımı

IV. YALANCI KODLAR VE AKIŞ DİYAGRAMI

1. Graf Oluşturma İşlemi

```
FUNCTION BuildGraph():

    cityData = LoadCityDataFromJSON()

    graph = new Graph()

    // Durakları graf yapısına ekleme

    FOR each stop IN cityData.stops:

        graph.addVertex(stop)

    // Kenarları (edge) ekleme: nextStops ve transfer bilgileri üzerinden

    FOR each stop IN cityData.stops:

        IF stop.hasNextStops():

            FOR each nextStopInfo IN stop.nextStops:

                targetStop = findStopById(nextStopInfo.stopId)

                IF targetStop EXISTS:

                    edge = new RouteEdge(nextStopInfo.distance, nextStopInfo.time, nextStopInfo.cost)

                    graph.addEdge(stop, targetStop, edge)

        IF stop.hasTransfer():

            transferStop = findStopById(stop.transfer.transferStopId)

            IF transferStop EXISTS:

                edge = new RouteEdge(0.0, stop.transfer.transferTime, stop.transfer.transferCost)
```

graph.addEdge(stop, transferStop, edge)

RETURN graph

2. Dijkstra Algoritması

FUNCTION Dijkstra(graph, start, end, weightType):

// Tüm düğümlerin mesafelerini sonsuz olarak başlat

FOR each vertex IN graph.vertices:

distance[vertex] = INFINITY

previous[vertex] = NULL

distance[start] = 0

// Öncelik kuyruğuna başlangıç düğümünü ekle

priorityQueue.add(start)

WHILE priorityQueue is NOT empty:

current = priorityQueue.removeMin()

IF current == end:

BREAK

FOR each edge IN graph.getEdges(current):

neighbor = edge.destination

weight = GetWeight(edge, weightType) // Ücret, süre veya mesafe seçeneğine göre

alternative = distance[current] + weight

IF alternative < distance[neighbor]:

distance[neighbor] = alternative

previous[neighbor] = current

priorityQueue.update(neighbor, distance[neighbor])

// Son düğüme ulaşılamadıysa boş yol döndür

IF distance[end] == INFINITY:

RETURN EMPTY_PATH

// Önceki düğümleri kullanarak yolu tersine oluştur

path = reconstructPath(previous, start, end)

RETURN path

3. Rota Hesaplamada Strateji Deseni Kullanımı

// RouteStrategy arayüzü: Tüm stratejiler bu arayüzü uygular

INTERFACE RouteStrategy:

FUNCTION calculateRouteHtml()

// En Uygun Ücretli Rota Stratejisi

CLASS CheapestRouteStrategy IMPLEMENTS RouteStrategy:

CONSTRUCTOR(rotaHesaplama)

FUNCTION calculateRouteHtml():

RETURN rotaHesaplama.getCheapestRouteHtml()

// En Uygun Zamanlı Rota Stratejisi

```
CLASS FastestRouteStrategy IMPLEMENTS RouteStrategy:

    CONSTRUCTOR(rotaHesaplama)

    FUNCTION calculateRouteHtml():

        RETURN rotaHesaplama.getFastestRouteHtml()

// Rota hesaplama için strateji seçimi fonksiyonu

FUNCTION getRouteHtml(strategyKey, rotaHesaplama):

    SWITCH(strategyKey.toLowerCase()):

        CASE "cheapest":

            strategy = new CheapestRouteStrategy(rotaHesaplama)

        CASE "fastest":

            strategy = new FastestRouteStrategy(rotaHesaplama)

        // Diğer strateji seçenekleri burada yer alır...

        DEFAULT:

            RAISE Exception("Unknown route strategy")

    RETURN strategy.calculateRouteHtml()
```

4. En Uygun Ücretli Rota Hesaplaması

```
FUNCTION getCheapestRouteHtml():

    graph = BuildGraph()

    // Başlangıç noktasına en yakın durağı belirle

    startSegment = processSegmentBetweenPointAndNearestStop(startLat, startLon, graph)

    // Hedefe en yakın durağı bul

    nearestDestination = findNearestStop(destLat, destLon, graph)

    // Dijkstra algoritması ile rota hesapla (ücret ağırlıklı)

    path = Dijkstra(graph, startSegment.stop, nearestDestination, "cost")

    // Rota detaylarını topla: mesafe, ücret, süre

    totalCost = startSegment.cost

    totalDistance = startSegment.distance

    totalTime = startSegment.time

    FOR each edge IN path:

        totalCost += edge.cost

        totalDistance += edge.distance

        totalTime += edge.time

    // Hedef segmenti (örneğin, taksiyle geçiş) ekle

    taxiSegment = processSegmentStopToPoint(nearestDestination, calculateDistance(nearestDestination, destinationPoint))

    totalCost += taxiSegment.cost

    totalDistance += taxiSegment.distance

    totalTime += taxiSegment.time

    // Hesaplanan rota detaylarını HTML formatında döndür
```

htmlOutput = formatRouteHtml(totalCost, totalDistance, totalTime, path)

RETURN htmlOutput

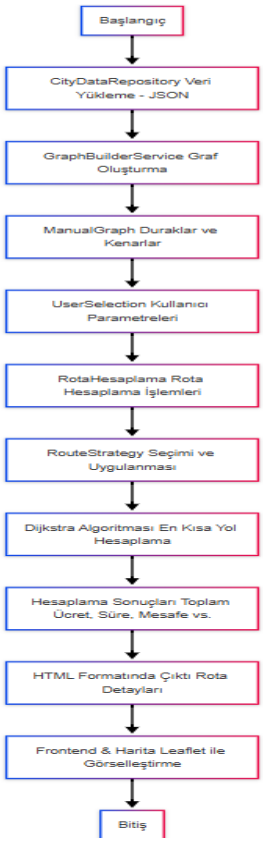

Fig.6 Akış Şeması

V. RAPORA KONULMASI İSTENEN SORULAR VE CEVAPLARI

1) Proje tamamlandıktan sonra, daha önce hiç kullanılmamış yeni bir ulaşım yöntemi (örneğin elektrikli scooter) ortaya çıktığında sisteme hangi değişiklikler yapılmalıdır?

Yeni Ulaşım Aracı Tanımlama: Mevcut yapıda, Arac adlı soyut (abstract) bir sınıf ve ondan türetilmiş (Otobüs, Tramvay, Taksi gibi) sınıflar bulunuyor. Yeni bir ulaşım aracı (örneğin Scooter) eklemek için, Arac sınıfını genişleten (extends Arac) yeni bir sınıf oluşturmak yeterli olacaktır. Bu sınıfta AracTipiGoster() gibi soyut metotları kendimize göre override edebilir, scooter için özel ücret/süre hesaplaması gerekiyorsa bunu ek metotlar aracılığıyla tanımlayabiliriz.

Rota Hesaplama ve Strateji: Scooter için farklı bir rota stratejisine ihtiyaç duyulursa RouteStrategy arayüzünü implemente eden yeni bir strateji sınıfı oluşturulabilir. Basit senaryolarda mevcut "en kısa" veya "en hızlı" stratejiler de kullanılabilir.

Veri Katmanı ve JSON Güncellemesi: Scooter durakları veya yolları JSON verisine eklenir. Eğer scooter durakları farklıysa, "type" alanıyla tanımlanarak GraphBuilderService'in ilgili düğümleri işlemesi sağlanır.

Factory (ObjectFactory) Güncellemesi: Kullanıcı arayüzünde "Scooter" seçeneği sunulacaksa, ObjectFactory'ye yeni bir kayıt (örneğin aracMap.put("scooter", Scooter::new)) eklenir.

Özetle, mevcut tasarım yeni sınıflar ve kayıtlar ekleyerek, kodda minimum değişiklikle yeni ulaşım araçlarını entegre etmeye uygundur.

2) Otonom taksi veya benzeri yeni ulaşım araçlarının projeye eklenmesi için ne tür değişiklikler gereklidir?

Var Olan Taksi Mantığını Genişletme: Mevcut Taxi sınıfından türeyen(extends Taxi) OtonomTaxi sınıfı oluşturulur. Bu sınıfta, örneğin daha düşük ücret veya sabit hız gibi farklı fiyatlandırma/süre hesaplama politikaları tanımlanabilir.

Rota Stratejisi ve Ödeme Yöntemi: Eğer otonom taksinin rota hesaplaması normal taksiyle aynıysa mevcut TaxiRouteStrategy kullanılabilir; aksi halde, yeni bir RouteStrategy (örneğin, AutonomousTaxiRouteStrategy) eklenir. Ödeme yöntemi tarafında özel bir durum yoksa, nakit/kredi kartı gibi

mevcut yöntemler aynı şekilde çalışmaya devam edecektir.

Arayüz (UI) ve Factory Entegrasyonu: Kullanıcı arayüzünde "Otonom Taksi" seçeneği sunulur ve ObjectFactory'ye "otonom_taxi", OtonomTaxi::new gibi bir kayıt eklenir.

Özetle, otonom taksi için yeni bir sınıf (ve gerekirse yeni bir strateji) eklenerek, mevcut yapıya minimum müdahale ile entegrasyon sağlanır.

3)Daha Önce Yazılmış Fonksiyonların Hangilerinde Değişiklik Yapılmalıdır?

Yeni ulaşım araçları (örneğin, scooter) ve otonom taksiler eklenirse, mevcut fonksiyonlarda yapılması gereken değişiklikler şunlardır:

ObjectFactory: Yeni araç, yolcu veya ödeme türleri için üretici (supplier) kayıtlarının eklenmesi gerekecektir.

GraphBuilderService: Yeni durak tipleri ve bağlantılar (örneğin, scooter istasyonları veya otonom taksi transferleri) işlenebilmesi için verilerin filtreleme koşulları ve eklemeleri güncellenmelidir.

RotaHesaplama: Yeni araçlara özel ücret ve süre hesaplama formülleri varsa, processSegmentBetweenPointAndNearestStop ve processSegmentStopToPoint gibi metotlarda bu farklı kriterler eklenmelidir.

TaxiSınıfı: Eğer otonom taksi, normal taksiden farklı hesaplama yapıyorsa, yeni bir sınıf oluşturulup (örneğin, OtonomTaxi) uygun metotlar override edilmelidir.

Bu değişiklikler, mevcut kodun yapısını bozmadan yeni işlevlerin eklenmesine olanak sağlayacaktır.

4) Açık/Kapalı Prensibi (Open/Closed Principle) bağlamında, mevcut fonksiyonlarda herhangi bir değişiklik yapmadan yeni ulaşım araçları sisteme nasıl entegre edilebilir? Nesne hiyerarşisi başlangıçta nasıl tanımlanmış olsaydı, yeni araçların eklenmesi daha kolay olurdu?

Open/Closed Prensibi Nedir?

Yazılım bileşenlerinin yeni davranışlar eklemek için açık, mevcut davranışları değiştirmek için kapalı olması gerektiğini söyler. Yani, var olan kodu mümkün olduğunca değiştirmeden, sistemi yeni özelliklerle genişletebilmeliyiz.

Mevcut Yapının Uygunluğu: Kodumuzda Arac soyut sınıfı, RouteStrategy arayüzü ve ObjectFactory gibi yapıların bulunması, yeni bir aracı eklemek istediğimizde yalnızca yeni bir sınıf ekleyerek (extends Arac) veya yeni bir strateji ekleyerek (implements RouteStrategy) işlevselliği genişletmemize olanak tanır.

Örneğin, elektrikli scooter eklemek için:

class Scooter extends Arac şeklinde yeni bir sınıf oluştururuz.

AracTipiGoster() metodunu scooter'a özel şekilde override ederiz.

Gerekirse ObjectFactory veya rota hesaplamalarında bu araca ait bir kayıt ekleriz.

Böylece, mevcut Otobüs/Tramvay/Taksi kodlarında bir değişiklik yapmaya gerek kalmaz.

Kod Değişikliği Minimize Edilir: Var olan sınıfları değiştirmek yerine, yeni sınıflar veya metotlar ekleyerek sistem genişletilir. Bu yaklaşım, **Açık/Kapalı Prensibi**ne tam uyum sağlar.

Eğer başlangıçta tüm araçların ortak özelliklerini barındıran soyut bir Arac sınıfı tanımlanıp(örn. AracTipiGoster() gibi), tüm

araçlar (Taksi, Otobüs, Tramvay vb.) bu sınıftan türetilmiş olsaydı, yeni araç eklemek sadece bu sınıftan miras alıp özel metotları override etmekle mümkün olurdu. Bu, sistemin genişletilebilirliğini artırırdı. Bu sayede, "sistemin kapısı" yeni alt sınıflara açık kalacak ama mevcut Arac veya RotaHesaplama içindeki kodu değiştirmek zorunda kalmayacağız.

5) 65 yaş ve üzeri bireyler için ücretsiz seyahat hakkının 20 seyahat ile sınırlandırılması gerektiğinde , bu değişiklik projeye sonradan nasıl eklenebilir?

65 yaş ve üzeri yolcular için ücretsiz seyahat hakkını 20 seferle sınırlandırmak için mevcut kodda büyük bir değişiklik yapmadan, yeni bir sınıf ekleyerek veya mevcut "Yasli" sınıfını genişleterek bu iş kuralını entegre edebilirsiniz. Örneğin:

Yeni Bir Alt Sınıf Oluşturma: "Yasli" sınıfını genişleterek, ücretsiz seyahat sayısını takip eden (örneğin bir **sayaç** alanı ekleyerek) ve 20 sefer sınırını kontrol eden bir "LimitedFreeYasli" sınıfı oluşturabilirsiniz. Bu sınıfta, IndirimUygula metodunu override ederek, ücretsiz seyahat hakkı varsa toplam ücreti 0 yapıp sayaç arttırılır, hak tükendiğinde ise normal indirim uygulanır.

Fabrika Güncellemesi: Yeni sınıfı, nesne üretiminde kullanmak üzere ObjectFactory'ye ekleyebilirsiniz. Böylece, proje genelinde diğer kodlarda değişiklik yapmadan bu iş kuralı sisteme entegre edilmiş olur.

Bu yaklaşım, mevcut kodu bozmadan yeni iş kurallarını eklemenizi sağlar.

6) Bu sınırlama kod üzerinde nasıl uygulanmalıdır? Hangi sınıf ve fonksiyonlar etkilenecektir?

Bu sınırlandırmayı uygulamak için öncelikle mevcut Yasli sınıfı etkilenir. Önerilen yaklaşım:

"Yasli" sınıfını genişleterek, örneğin LimitedFreeYasli adında bir alt sınıf oluşturulur. Bu sınıfta bir sayaç eklenir ve her ücretsiz seyahat hakkı kullanıldığında sayaç artar.

IndirimUygula() metodunu override ederek, ücretsiz seyahat hakkı (20 sefer) henüz geçerli ise ücreti 0 yapar; hak tükendiğinde normal indirim uygulanır.

ObjectFactory: Yeni sınıfı üretmek üzere güncellenir. Böylece, ilgili yolcu tipi seçildiğinde LimitedFreeYasli örneği oluşturulur.

RotaHesaplama: Rota hesaplamalarında, ücretsiz seyahat indirimi uygulanan yolcu bilgisi kullanıldığı için, calculateAdjustedCost() gibi fonksiyonlar bu değişiklikten dolaylı olarak etkilenir.

Bu şekilde, mevcut kod yapısını büyük ölçüde değiştirmeden yeni iş kuralı entegre edilebilir.

VI. DENEYSEL SONUÇLAR

Projede, rota planlama sisteminin farklı senaryolar altındaki performansı kapsamlı deneysel testlerle değerlendirilmiştir. Kısaca:

Rota Hesaplamaları: Dijkstra algoritması kullanılarak, farklı ağırlık kriterleriyle (ücret, süre, mesafe) en uygun rota başarıyla hesaplanmıştır. Otobüs, tramvay ve taksi için ayrı grafik yapıları oluşturulmuş; taksi hesaplamalarında ödeme yöntemi de dikkate alınmıştır.

Performans ve Hata Yönetimi: Hesaplamalar düşük gecikmeyle gerçek zamanlı sonuçlar sunmuş; rota bulunamadığında bilgilendirici hata mesajları verilmiş ve JSON

veri okuma, graf oluşturma gibi aşamalarda sağlam hata yönetimi sağlanmıştır.

Kullanıcı Etkileşimi: Leaflet tabanlı harita görselleştirmesi ve dinamik HTML çıktıları, kullanıcıya interaktif bir deneyim sunarak rota detaylarını (mesafe, ücret, süre) açıkça göstermiştir.

Özelleştirme ve Genel Değerlendirme: Yolcu tipleri ve ödeme yöntemlerinin etkileri net bir şekilde gözlemlenmiş, sistemin esnekliği ve algoritmik doğruluğu ortaya konulmuştur. Deneysel sonuçlar, sistemin hem teknik performans hem de kullanıcı deneyimi açısından başarılı olduğunu göstermiştir.

VII. SONUÇ

Proje, kullanıcıların başlangıç ve varış noktalarını seçerek ücret, süre ve mesafe kriterlerine göre en uygun rotayı hesaplayan dinamik ve modüler bir sistem sunmaktadır. Nesne yönelimli programlama, strateji deseni ve graf teorisine dayalı algoritmalar sayesinde otobüs, tramvay ve taksi gibi farklı ulaşım modları entegre edilmiştir. Deneysel sonuçlar, sistemin yüksek performans, doğruluk ve kararlılık sağladığını; indirim, ek ücret ve transferlerin doğru hesaplandığını göstermektedir. İnteraktif harita entegrasyonu ve kullanıcı dostu arayüz, sistemi pratik kılarken, modüler yapısı gelecekteki iyileştirmelere de olanak tanımaktadır.

VIII. KAYNAKÇA

https://academy.patika.dev/courses/java-spring-boot/spring-boot-nedir

https://metintopcu1.medium.com/spring-vs-spring-boot-nedir-badad8ef7c72

https://coderspace.io/en/blog/what-is-solid-examples-of-solid-principles/

https://academy.patika.dev/courses/java-spring/maven-nedir

https://spring.io/projects/spring-boot

https://www.openstreet-map.org/#map=6/39.03/35.25

https://medium.com/kodcular/api-nedir-1eb5ea2faf44

https://www.patika.dev/blog/api-nedir-api-ne-ise-yarar

https://www.yusufsezer.com.tr/spring-web-mvc/

https://codegym.cc/tr/gro-ups/posts/tr.232.solid-javada-snf-tasarmnn-bes-temel-ilkesi