DERS ADI Görsel Programlama II
ÜNİTE ADI JAVAFX İLE UYGULAMA GELİŞTİRMEYE GEÇİŞ
ÜNİTE NO 1
YAZAR EMİR SULTAN GÖKKAYA

Bu ünitede sizlerle birlikte, JavaFX ile arayüz bileşenlerini tanıma aşamasından uygulama geliştirme düzeyine geçişi sistemli biçimde ele almaktayız. Görsel Programlama I'de Stage–Scene–Node ilişkisini, temel bileşenleri ve olay yönetimini kullanarak ilk etkileşimli ekranlarımızı geliştirmiştik. Ancak gerçek bir masaüstü uygulaması geliştirmek, yalnızca ekrana buton yerleştirmekten ibaret değildir. Kullanıcı eylemini iş akışına dönüştürmek, arayüz–veri ilişkisini tutarlı biçimde yönetmek ve proje yapısını büyüdükçe bozulmayacak şekilde örgütlemek gerekir. Bu nedenle bu ünitenin temel amacı, sizlere JavaFX'te sürdürülebilir bir başlangıç iskeleti kazandırmaktır.
Ünite boyunca şu sorulara uygulama odaklı yanıt üretmekteyiz: NetBeans üzerinde JavaFX projesi nasıl oluşturulur? Proje yapısı nasıl okunur? JavaFX yaşam döngüsü neden önemlidir? Tek ekranlı bir arayüz iskeleti nasıl kurulur? Ayrıca JDK 11 ve sonrası sürümlerde JavaFX'in JDK'den ayrışmış olması nedeniyle, ortam uyumluluğu ve şablon görünürlüğü gibi pratik konuların da erken aşamada bilinmesi gerektiğini vurgulamaktayız. Böylece sizler, yalnızca "çalışan bir ekran" değil, sonraki ünitelerin üzerine inşa edileceği sağlam bir temel kurmayı öğrenmektesiniz.

NetBeans IDE ile JavaFX Projesi Oluşturma (Java with Ant)

Bu kitapta örnek uygulamaları NetBeans IDE üzerinde, Java with Ant > JavaFX Application şablonuyla geliştirmekteyiz. Bu tercih, özellikle ders ortamında hızlı başlangıç ve herkesin benzer bir proje iskeletiyle ilerlemesi açısından önemli avantaj sağlamaktadır. Ancak burada kritik bir teknik not bulunmaktadır: JDK 11 ve sonrasında JavaFX, JDK'den ayrılmış bağımsız bir bileşen hâline geldiği için, Ant tabanlı JavaFX şablonlarının görünmesi ya da sorunsuz çalışması kullanılan ortama göre değişebilmektedir. Bu nedenle proje oluşturma aşamasında IDE–JDK–JavaFX uyumluluğunu dikkate almak, ileride karşılaşılabilecek yapılandırma sorunlarını azaltır.
Bu başlık altında amacımız yalnızca "proje açmak" değil, uygulama geliştirme sürecini standartlaştırmak ve sınıf içinde ortak bir başlangıç noktası oluşturmaktır. Böylece hem anlatılan örnekler daha kolay takip edilir hem de öğrenciler arasında proje yapısı kaynaklı farklılıklar azalır.

Proje Sihirbazını Açma

İlk adımda NetBeans içinde File > New Project yoluyla proje sihirbazına erişmekteyiz. Bu adım görünüşte basit olsa da pedagojik açıdan önemlidir. Çünkü uygulama geliştirme sürecinin düzenli ve tekrar üretilebilir olması, doğru başlangıç adımlarının alışkanlık hâline gelmesiyle mümkündür. Dokümanda bu aşama ekran görüntüsüyle desteklenerek öğrencinin doğru menü yolunu doğrulaması sağlanmaktadır. Özellikle sayfa 3'teki görsel, başlangıç ekranından "New Project" erişimini açık biçimde göstermektedir.

JavaFX Şablonunu Seçme (Java with Ant)

Proje sihirbazı açıldıktan sonra Categories bölümünden Java with Ant, Projects bölümünden ise JavaFX Application seçilmektedir. Bu seçim, sonraki örneklerde kullanılacak başlangıç dosyalarının ve proje yapısının temelini oluşturduğu için kritik bir eşiktir. Yanlış şablon seçimi, kitapta izlenen akıştan kopmaya neden olabileceğinden bu adım özellikle vurgulanmaktadır. Sayfa 4'teki ekran görüntüsü, öğrencilerin doğru kategori ve proje türünü görsel olarak eşleştirmesine yardımcı olmaktadır.

Proje Adı ve Konumlandırma

Bir sonraki aşamada proje adı, konumu ve temel ayarlar belirlenmektedir. Burada yalnızca teknik bir form doldurma işlemi yapılmamaktadır. Aynı zamanda okunabilir ve düzenli bir proje adlandırma alışkanlığı kazandırılmaktadır. Örneğin Unit01_Starter gibi kısa ve anlamlı adlar, dönem boyunca birden fazla örnek proje üretildiğinde dosya yönetimini kolaylaştırır. Bu yaklaşım, akademik ve kurumsal çalışma disiplinine de uygundur. Sayfa 5'te yer alan "Name and Location" ekranı bu süreci somutlaştırmaktadır.

Proje Yapısını Tanıma: "Uygulama İskeleti" Nerede?

Proje oluşturulduktan sonra NetBeans'in otomatik kurduğu klasör ve dosya yapısı incelenmektedir. Bu noktada hedefimiz, "hangi dosya ne işe yarar?" sorusuna hızlı ve işlevsel bir yanıt verebilmektir. Çünkü başlangıçta proje yapısını anlamadan yalnızca kod yazmaya odaklanmak, ilerleyen aşamalarda bakım sorunlarına neden olur. Uygulama iskeleti kavramı tam da burada devreye girmektedir: Amaç, küçük bir örneği bile büyümeye uygun bir mantıkla başlatmaktır. Sayfa 5'teki Projects paneli görünümü, bu yapının NetBeans üzerinde nasıl göründüğünü öğrencinin zihninde netleştirir.
Dokümanda ayrıca JavaFX uygulamasının giriş noktası olan sınıfın çoğunlukla Application sınıfını genişletecek şekilde oluşturulduğu belirtilmektedir. Bu bilgi, proje yapısını yalnızca dosya listesi

olarak değil, çalışma zamanı mantığıyla ilişkili bir yapı olarak görmemizi sağlar.

İlk Çalıştırma ve UI Çıktısı

Bu ünitede kod anlatımında görselleştirilen ana unsur, kodun kendisi değil kodun ürettiği UI çıktısıdır. Bu yaklaşım, özellikle başlangıç düzeyindeki öğrenciler için çok değerlidir. Çünkü uzun kod bloklarını ezberlemektense, "doğru ilerlediğimde ekranda ne görmeliyim?" sorusuna cevap vermek öğrenmeyi hızlandırır. Projeyi Run Project komutuyla çalıştırıp ilk pencereyi görmek, doğru yapılandırmanın hızlı bir doğrulamasıdır. Sayfa 6'daki pencere çıktısı (Hello World benzeri ekran), bu doğrulama yaklaşımının görsel karşılığıdır.

JavaFX Uygulama Yaşam Döngüsü: "Uygulama Geliştirme" Bakışı

JavaFX'te uygulamanın giriş noktası javafx.application.Application sınıfını genişleten bir sınıftır. Burada önemli olan nokta şudur: Uygulamayı doğrudan geliştirici değil, JavaFX çalışma zamanı oluşturur ve yönetir. Bu nedenle yaşam döngüsünü doğru anlamak, uygulamanın ne zaman hazırlandığını, ne zaman arayüz kurduğunu ve ne zaman kapandığını kontrol etmek açısından kritik önemdedir. Dokümanda yaşam döngüsünün init(), start(), stop() adımlarından oluştuğu açık biçimde belirtilmektedir.

Yaşam Döngüsü Adımları

init(): Sahne henüz gösterilmeden önce temel hazırlıklar için kullanılır.
start(Stage primaryStage): Arayüzün kurulduğu, Scene ile Stage ilişkisinin yapıldığı ana aşamadır.
stop(): Uygulama kapanırken kaynakların güvenli biçimde serbest bırakıldığı aşamadır.
Bu sıralamayı yalnızca teorik bilgi olarak değil, "hazırlık–arayüz kurma–kapanışta temizlik" düzeni olarak kavramanız gerekmektedir. Çünkü ilerleyen ünitelerde dosya, veritabanı veya arka plan süreçleri gibi kaynaklarla çalışırken bu düzen doğrudan önem kazanacaktır.
Dokümanda ayrıca JavaFX'in arayüz güncellemelerini yürütmek için ayrı bir JavaFX Application Thread oluşturduğu ve Scene graph üzerindeki canlı nesne değişikliklerinin bu iş parçacığında yapılması gerektiği vurgulanmaktadır. Bu ilke, UI tarafında donma veya tutarsız güncellemelerin önlenmesi açısından temel bir kuraldır.

Uygulama Kapatma Mantığı: Platform.exit()

Uygulamanın kontrollü sonlandırılmasında Platform.exit() çağrısı önemli bir araçtır. Bu çağrı, uygun koşullarda stop() metodunun devreye girmesini ve JavaFX uygulama iş parçacığının sonlanmasını sağlar. Ders bağlamında bu yaklaşımın önemi, "kapatırken veri kaybı" ya da kapanışta temizlenmeyen kaynaklar gibi sorunların önüne geçmesidir. Yani kapatma işlemi, yalnızca pencereyi kapatmak değil, uygulamayı düzenli biçimde sonlandırmak olarak ele alınmaktadır.

"Tek Ekran" İçinde Uygulama İskeleti Kurmak: Kod Tabanlı UI

Bu kitapta FXML ve Scene Builder daha sonraki ünitelerde ele alınacağı için, Ünite 1'de arayüzü programatik (kod tabanlı) biçimde kurmaktayız. Bu tercih pedagojik olarak bilinçlidir. Çünkü başlangıç aşamasında JavaFX API'nin mantığını doğrudan görmek, sahne grafiğinin nasıl oluştuğunu anlamayı kolaylaştırır. Label, Button, VBox, Scene ve Stage ile kurulan basit bir ekran iskeleti, JavaFX'in temel yapı taşlarının nasıl bir araya geldiğini açık biçimde göstermektedir. Dokümandaki örnek kodda sahne kurulumunun start() içinde yapılması, JavaFX'in temel uygulama yapısıyla uyumlu yaklaşımı temsil etmektedir.

Paketleme: model – service – ui

Projeler büyüdükçe her şeyi tek sınıfta toplamak bakım maliyetini artırır. Bu nedenle dokümanda, daha ilk üniteden itibaren model – service – ui ayrımı önerilmektedir.
model: Alan/varlık nesneleri
service: In-memory işlemler (ekle, sil, listele vb.)
ui: JavaFX arayüzünü kuran sınıflar
Bu ayrım, tek sorumluluk ilkesini öğrenciye pratik düzeyde göstermekte ve ileride daha büyük projelerde kod yığılmasını azaltacak bir alışkanlık kazandırmaktadır.

Başlangıç İskeleti: start() içinde sahne kurulumu

Ünitede verilen başlangıç iskeleti örneği, start() içinde Scene ve Stage kurulumunun nasıl yapıldığını göstermektedir. Buradaki amaç tam işlevsel bir uygulama üretmekten çok, JavaFX uygulamasının " çalışan iskeletini" doğru biçimde kurmaktır. Öğrenci bu iskelet sayesinde, arayüzün rastgele değil, belirli bir yaşam döngüsü ve sahne grafiği mantığı içinde oluştuğunu kavrar.

Ünite 02'ye Köprü: "Observable" Düşüncesi Neden Gerekli?

Bu ünitede observable yapılar ayrıntılı işlenmemekle birlikte, neden gerekli oldukları kavramsal olarak temellendirilmektedir. Kullanıcı arayüzleri değişen veriyi göstermek zorundadır. Eğer veri sıradan bir ArrayList ile tutulursa, arayüz bu değişimi otomatik olarak fark etmez ve geliştirici manuel güncelleme yapmak zorunda kalır. Dokümanda özellikle "ArrayList UI'yi otomatik güncellemez" vurgusu yapılarak bu sorunun altı çizilmektedir. Ünite 2'de ObservableList ve seçim kontrolleri ele alınırken, bu yükün nasıl azaltıldığı uygulamalı biçimde gösterilecektir.

DERS ADI Görsel Programlama II
ÜNİTE ADI GELİŞMİŞ ARAYÜZ BİLEŞENLERİ
ÜNİTE NO 2
YAZAR EMİR SULTAN GÖKKAYA

Bu ünitede sizlerle, masaüstü uygulamalarında kullanıcı etkileşimini belirleyen temel unsurun çoğu zaman seçim davranışları olduğunu ele almaktayız. Bir öğeyi seçmek, bir modu belirlemek, bir özelliği açıp kapatmak ya da sağ tık menüsünden işlem başlatmak yalnızca teknik birer tıklama değildir. Bunlar, uygulamanın "hangi nesne üzerinde", "hangi amaçla" ve "hangi durumda" çalışacağını kullanıcı adına tanımlayan etkileşimlerdir. Bu nedenle seçim kontrolleri, kullanıcı deneyiminde hız, doğruluk ve kontrol hissi bakımından belirleyici bir role sahiptir. Ünite 1'de arayüzün görünür iskeletini kurmuştuk. Bu ünitede ise arayüzü statik bir ekran olmaktan çıkarıp, kullanıcı eylemlerini anlamlandıran bir etkileşim sistemine dönüştürmekteyiz.
Bu yaklaşım, doğrudan manipülasyon anlayışıyla da uyumludur. Kullanıcı arayüzdeki nesneler üzerinde doğrudan işlem yapar, sistem de bu işlemlerin sonucunu gecikmeden görünür kılar. JavaFX'in liste, seçim modeli, property ve listener yapıları bu davranışı uygulama düzeyinde kurmamıza yardımcı olur. Dolayısıyla bu ünitede yalnızca "hangi bileşen ne işe yarar" sorusunu değil, "seçim uygulama davranışına nasıl dönüştürülür" sorusunu da merkezde tutmaktayız.

ObservableList ile Veri Akışı: "Liste Değişince Arayüz Neden Güncelleniyor?"

JavaFX'te birçok kontrol veri kaynağını yalnızca okumaz, aynı zamanda veri değiştiğinde görünümün güncellenmesini bekler. Bu gereksinimi karşılayan temel yapı ObservableList'tir. ObservableList'i, değişiklikleri dinleyicilere haber veren bir liste olarak düşünebilirsiniz. Listeye yeni bir öğe eklendiğinde, bir öğe silindiğinde ya da sıralama değiştiğinde, bu değişim ilgili kontrol tarafından algılanır ve arayüz görünümü güncellenir. Bu, özellikle ListView ve benzeri kontrollerle çalışırken geliştiricinin manuel yenileme yükünü ciddi ölçüde azaltır.
Burada önemli olan nokta, veri güncelleme ile arayüz güncelleme arasındaki bağın daha tutarlı hâle gelmesidir. Siz veri modelini değiştirirsiniz, bağlı kontrol bu değişimi izler ve arayüzü uyumlu biçimde yeniden üretir. Bu yaklaşım, olay güdümlü programlama mantığıyla da örtüşür. Bir değişim " olay" olarak algılanır ve sistem buna tepki verir. Böylece özellikle büyüyen projelerde "veri değişti ama ekran güncellenmedi" türü tutarsızlıkların önüne geçmek kolaylaşır. ObservableList her sorunu tek başına çözmez, ancak veri–arayüz ilişkisinin temelini sağlamlaştırır.

ListView: Listeleme + Seçim Yönetiminin Merkezi

ListView'in Temel Mantığı

ListView, kaydırılabilir bir öğe listesi sunmanın yanında seçim yönetimini de bünyesinde taşıyan merkezi bir kontroldür. Bu nedenle ListView'i iki katmanlı düşünmek öğretici bir yaklaşımdır. Birinci katman veri katmanıdır ve çoğunlukla ObservableList ile tutulan öğeler items üzerinden kontrola bağlanır. İkinci katman ise seçim katmanıdır ve SelectionModel üzerinden kullanıcının hangi öğeyi seçtiği izlenir. Veri ile seçimi ayrı yönetmek, gerçek uygulamalarda davranışı daha öngörülebilir hâle getirir. Çünkü veri değişebilirken seçim aynı kalabilir ya da seçim değişirken veri kümesi sabit kalabilir.
ListView'in bir diğer güçlü yönü, hücre (cell) mantığı sayesinde görünümün gerektiğinde özelleştirilebilmesidir. Böylece yalnızca metin listeleri değil, ikonlu, etiketli ya da daha zengin görünümler de üretilebilir. Ancak bu ünitede temel odak, seçim mantığını doğru kurmaktır. Bu nedenle önce sade listeleme ve seçim akışını netleştirmekteyiz.

Seçim Olaylarını Yakalama

ListView'de seçim yönetimi çoğunlukla getSelectionModel() üzerinden yapılır. Buradan seçili öğe, seçili indeks ve seçili öğeler gibi bilgilere erişilebilir. Özellikle selectedItemProperty() üzerinde dinleyici tanımlamak, seçim değişimini anlık olarak yakalamanın en yaygın ve etkili yoludur. Seçim değiştiğinde butonların aktif/pasif duruma geçirilmesi, seçilen öğeye ait detay bilgisinin gösterilmesi veya bağlama göre eylemlerin görünür kılınması gibi davranışlar bu yapı üzerinden çalışır.
Buradaki temel tasarım ilkesi şudur: Seçim yalnızca teknik bir bilgi değildir. Seçim, uygulamanın o anki durumunu belirleyen bir sinyaldir. Kullanıcı arayüzü bu sinyale hızlı ve anlaşılır biçimde tepki verdiğinde, etkileşim akışı daha doğal ve güvenilir hâle gelir. Bu da ünitenin ana hedeflerinden biri olan "seçimin arayüz davranışını değiştirmesi" ilkesini somutlaştırır.

Tekli ve Çoklu Seçim

ListView'in seçim modeli, hem tekli hem de çoklu seçimi destekleyecek biçimde tasarlanmıştır.

Varsayılan davranış genellikle tekli seçimdir. Bu tercih, yanlışlıkla çoklu seçim yapılmasını azaltır. Ancak uygulama gereksinimi birden fazla öğe üzerinde işlem yapmayı gerektiriyorsa, seçim modu bilinçli biçimde SelectionMode.MULTIPLE olarak ayarlanmalıdır. Böylece kullanıcı birden fazla kayıt seçip toplu silme, etiketleme veya başka toplu işlemler yapabilir.

Burada dikkat edilmesi gereken nokta, çoklu seçimin teknik olarak etkinleştirilmesinin yanında kullanıcıya bu davranışın nasıl çalıştığının da sezdirilmesidir. Arayüz tasarımında seçim davranışını açık ve öngörülebilir kılmak, yalnızca kod doğruluğu değil, kullanılabilirlik açısından da önem taşır.

ChoiceBox ve ComboBox Bileşenleri
Bu bölümde, seçim kontrollerini gereksinime göre doğru seçmenin kullanıcı deneyimine etkisini ele almaktayız. ChoiceBox, küçük ve önceden tanımlı seçenek kümeleri için sade ve hızlı bir çözümdür. Kullanıcı burada tek bir seçim yapar ve arayüzde gereksiz karmaşıklık oluşmaz. "Öncelik: Düşük / Orta / Yüksek" gibi sınırlı seçenekli senaryolarda ChoiceBox son derece uygundur.
ComboBox ise seçenek sayısı arttığında veya daha esnek bir yapı gerektiğinde öne çıkar. Açılır liste mantığıyla çalışır ve observable veriyle birlikte kullanıldığında veri değişimlerine uyumlu bir seçim deneyimi sağlar. Dolayısıyla ChoiceBox ile ComboBox arasındaki fark yalnızca görünüş farkı değildir. Bu fark, bilişsel yükü azaltma, kullanıcı hatasını önleme ve arayüz akışını sadeleştirme açısından bir tasarım kararıdır. Sizden beklenen, her iki kontrolü de ezberlemek değil, hangi senaryoda hangisinin daha uygun olduğunu ayırt edebilmektir.

Slider ve Spinner Bileşenleri
Sayısal tercih toplama senaryolarında metin kutusuna sayı yazdırmak çoğu zaman daha fazla hata üretir. Bu nedenle JavaFX'te Slider ve Spinner gibi kontroller daha güvenli ve kullanıcı dostu seçenekler sunar. Slider, kullanıcıya belirli bir aralıkta hızlı ve sezgisel ayar yapma imkânı verir. Değerin sürükleme ile değişmesi ve anlık geri bildirim alınabilmesi, özellikle yaklaşık seçimlerde avantaj sağlar.
Spinner ise adım adım artırma-azaltma mantığıyla daha hassas ve kontrollü seçimlerde öne çıkar. Burada özellikle SpinnerValueFactory yapısı, arayüz ile değer üretim kuralını birbirinden ayırdığı için öğretici bir örnektir. Aralık, başlangıç değeri ve artış miktarı gibi kurallar bu yapı üzerinden tanımlanır. Böylece geliştirici yalnızca arayüzü değil, sayı üretim davranışını da düzenli biçimde yönetebilir. Bu ayrım, ilerleyen aşamalarda bakım ve genişletme açısından önemli kolaylık sağlar.

ContextMenu (Sağ Tık) Bileşeni
ContextMenu, sağ tıklama ile açılan ve çoğunlukla MenuItem öğelerinden oluşan bağlama duyarlı bir menüdür. Bu yaklaşımın en önemli avantajı, arayüzü kalabalıklaştırmadan işlev sunabilmesidir. Kullanıcı tüm eylemleri sürekli ekranda görmek zorunda kalmaz. Bunun yerine yalnızca ihtiyaç duyduğu anda, ilgili öğe üzerinde işlem yaparken menü açılır. Bu, masaüstü uygulamalarında bağlama göre eylem sunmanın yaygın ve etkili bir yoludur.
Bu ünitede verilen örnekte, ListView üzerinde sağ tıklama ile "Sil" ve "Düzenle" işlemleri yapılmaktadır. Menü eylemlerinin seçili öğe üzerinden çalışması, seçim modeli ile bağlama duyarlı menü arasındaki ilişkiyi açık biçimde göstermektedir. Kullanıcı seçimi yapar, ardından o seçime uygun eylemleri görür ve doğrudan işlem başlatır. Bu akış, doğrudan manipülasyon yaklaşımının grafik arayüzü düzeyindeki karşılıklarından biridir. Ayrıca arayüz öğrenilebilirliğini artırır. Çünkü kullanıcı, ilgili eylemleri ihtiyaç anında ve doğru bağlamda görür.

DERS ADI Görsel Programlama II
ÜNİTE ADI TABLO TABANLI VERİ GÖSTERİMİ
ÜNİTE NO 3
YAZAR EMİR SULTAN GÖKKAYA

Bu ünitede sizlerle birlikte, masaüstü uygulamalarda çok alanlı verilerin tablo tabanlı biçimde nasıl sunulacağını sistematik olarak ele almaktayız. Günlük hayatta kullandığımız uygulamalarda veriler çoğu zaman tek satırlı değildir. Bir öğrencinin numarası, adı soyadı, bölümü, notu ve durumu gibi birden fazla özelliği aynı kayıt içinde yer almaktadır. Benzer biçimde sipariş, stok, personel ve ders kayıtları da çok sütunlu bir yapı gerektirmektedir. Bu nedenle yalnızca tek satırlı listeleme yaklaşımı çoğu durumda yetersiz kalmaktadır. İşte bu noktada JavaFX'in TableView bileşeni devreye girmekte ve veriyi satır-sütun mantığıyla düzenli, okunabilir ve etkileşimli biçimde sunmaktadır.

Bu ünitede önceki ünitede öğrendiğiniz ObservableList yaklaşımını tablo bağlamında genişletmekteyiz. Çünkü TableView de veriyi liste mantığıyla yönetmektedir, ancak aynı kaydın farklı özelliklerini sütunlara ayırarak kullanıcıya çok boyutlu bir okuma imkânı sağlamaktadır. Bunun yanında yalnızca gösterim değil, sıralama, filtreleme, seçim ve temel satır işlemleri gibi işlevler de tablo yönetiminin ayrılmaz parçalarıdır. Bu nedenle ünitenin genel hedefi, gerçek hayatta sık karşılaşacağınız "tablo yönetimi" senaryosunu gereksiz karmaşıklık oluşturmadan çalışır hâle getirmektir. Bu aşamada veri tabanı veya dosya kalıcılığına girmeden, arayüz ve veri bağlama mantığını sağlamlaştırmaya odaklanmaktayız.

TableView Ne Zaman Tercih Edilir?
Bir arayüzde hangi kontrolün seçileceği yalnızca estetik bir karar değildir. Bu, doğrudan kullanıcı verimliliğini belirleyen tasarım kararıdır. Eğer kullanıcı kayıtlar arasında karşılaştırma yapacaksa, sütun başlıklarına göre sıralama isteyecekse, satır seçerek detay görme veya silme gibi işlemler yapacaksa, TableView çoğu zaman en uygun seçim olmaktadır. Çünkü her satır bir varlığı, her sütun ise o varlığın bir özelliğini temsil etmektedir. Bu yapı, kayıt temelli düşünmeye son derece uygundur.

Buna karşılık ListView daha çok tek boyutlu liste mantığına yakındır. Çok alanlı kayıtlar ListView içinde gösterildiğinde kullanıcı ya uzun metin satırlarını okumak zorunda kalmakta ya da sürekli detay ekranlarına yönlendirilmektedir. Bu da etkileşimi yavaşlatmaktadır. Eğer kullanıcı bir kaydı anlamak için sürekli başka ekrana geçmek zorunda kalıyorsa, çoğu durumda çözüm kayıtların temel alanlarını doğrudan tabloda göstermektir. Bu nedenle TableView tercihi, sadece teknik değil, aynı zamanda kullanıcı deneyimi odaklı bir karardır.

Temel Yapı Taşları: TableView, TableColumn, TableCell
Bu bölümde tablo yapısını oluşturan üç temel sınıfı birlikte netleştirmekteyiz: TableView, TableColumn ve TableCell. Tablonun mantığını doğru kavramak için bu üç yapının görevlerini birbirinden ayırarak düşünmeniz gerekmektedir. Çünkü çoğu hata, bileşenlerin görevlerini karıştırmaktan kaynaklanmaktadır.

TableView: Tabloyu Taşıyan Kontrol
TableView, tablonun ana gövdesidir. Hangi verinin gösterileceği items mantığıyla belirlenmektedir. Bu nedenle geliştirme sürecinde ilk soru her zaman "Bu tablo hangi listeyi gösterecek?" sorusu olmalıdır. TableView yalnızca veriyi göstermekle kalmaz, aynı zamanda seçim yönetimi için SelectionModel sağlar. Tekli ya da çoklu seçim, seçili satırın alınması ve seçim durumuna göre işlem tetiklenmesi gibi senaryolar bu yapı üzerinden yürütülmektedir. Ayrıca düzenlenebilir tablo senaryolarında editable özelliğinin tablo, sütun ve hücre düzeyinde birlikte doğru yapılandırılması gerekmektedir. Öğrencilerin sık yaşadığı "Tablom edit olmuyor" problemi çoğunlukla bu koşullardan birinin eksik bırakılmasından doğmaktadır.

TableColumn: Sütunlar ve Cell Value Factory Mantığı
Her TableColumn, satırdaki verinin belirli bir özelliğini temsil etmektedir. Yani sütun, "Bu alanda hangi değer gösterilecek?" sorusunun cevabıdır. Bu cevap cellValueFactory aracılığıyla verilmektedir. Buradaki mantık yalnızca bir değer döndürmek değil, mümkün olduğunca gözlemlenebilir bir değer döndürerek değişimlerin arayüze yansıtılmasını sağlamaktır. İşte bu nedenle JavaFX property yaklaşımı tablo bileşenleriyle çok uyumlu çalışmaktadır.

Başlangıç düzeyinde PropertyValueFactory hızlı kurulum için önemli bir kolaylık sunmaktadır. Ancak property adları metin olarak verildiği için yazım hataları sütunun boş görünmesine neden olabilmektedir. Bu yüzden öğretim açısından şu dengeyi kurmak önemlidir: İlk denemelerde PropertyValueFactory ile mantığı görmek faydalıdır; daha kontrollü ve sürdürülebilir kurulum için ise property döndüren lambda bağlamaları tercih edilmelidir. Böylece hem okunabilirlik artmakta hem de hata ayıklama süreci kolaylaşmaktadır.

TableCell: Hücrenin Çizimi ve Düzenlenmesi
TableCell, satır ve sütunun kesişimidir. Yani ekranda gördüğünüz her hücre, aslında belirli bir satır indeksine ve belirli bir sütun bağlamına sahiptir. Hücre mantığını anlamak, tabloyu yalnızca gösterim aracı olarak değil, düzenlenebilir bir etkileşim alanı olarak kavramanızı sağlamaktadır. Özellikle düzenlenebilir sütunlarda TextFieldTableCell gibi hazır hücre sınıfları kullanılabilmektedir. Ancak burada girilen metnin veri modeline geri aktarılması için dönüştürücü (StringConverter) mantığının gerekli olduğu unutulmamalıdır. Aksi durumda kullanıcı metni değiştirir, fakat bu değişiklik veri modeline doğru türde aktarılamaz.

Örnek Uygulama – Öğrenci Not Tablosu
Bu ünitede kavramları soyut düzeyde bırakmamak için "Öğrenci Not Tablosu" uygulaması üzerinden ilerlemekteyiz. Uygulamanın amacı, tek ekranlı ve sade bir senaryo içinde tablo yönetiminin çekirdek işlevlerini kurmaktır. Bu işlevler öğrenci kaydı ekleme, çok sütunlu gösterim, arama ile filtreleme, sütun başlığına göre sıralama ve seçili satırı silme işlemlerinden oluşmaktadır. Girdi doğrulama bu aşamada minimum düzeyde tutulmakta, ayrıntılı doğrulama ise daha sonraki üniteye bırakılmaktadır. Bu yaklaşım pedagojik olarak doğrudur; çünkü aynı anda çok fazla kural eklemek, öğrencinin temel tablo mantığını kavramasını zorlaştırmaktadır.

Önce proje yapısı düzenlenmekte, ardından model ve ui paketleri ayrılarak kodun okunabilirliği artırılmaktadır. StudentRecord model sınıfında öğrenciye ait alanlar JavaFX property yapısıyla tanımlanmakta ve ayrıca not alanına göre hesaplanan bir getStatus() metodu ile "Geçti/Kaldı" bilgisi üretilmektedir. Böylece öğrenciler veri modelinde hem saklanan alanı hem de hesaplanan alanı nasıl yöneteceklerini görmektedir. Bu, tablo tasarımında çok önemli bir adımdır. Çünkü gerçek uygulamalarda bazı sütunlar doğrudan veri tabanından gelmez; uygulama mantığı içinde hesaplanarak gösterilir.

Arayüz tarafında giriş alanları, bölüm seçimi için ComboBox, not seçimi için Spinner, kayıt ekleme ve satır silme butonları ile arama kutusu oluşturulmakta; alt bölümde çok sütunlu TableView tanımlanmaktadır. Sütunlar No, Ad Soyad, Bölüm, Not ve Durum biçiminde kurulmakta, her biri uygun property ya da hesaplanan değerle bağlanmaktadır. Bu noktada özellikle "Durum" sütununun cellValueFactory içinde string üretimiyle verilmesi, hesaplanan sütun mantığını göstermesi açısından öğretici bir örnektir.

Tabloyu Veriyle Besleme: ObservableList ve Demo Veriler
TableView'in yalnızca sütunlarını tanımlamak yeterli değildir. Tablonun çalışabilmesi için veri listesi ile bağlanması gerekmektedir. Bu nedenle FXCollections.observableArrayList() ile üretilen ObservableList, tablonun veri kaynağı olarak kullanılmaktadır. Demo veriler eklenerek uygulamanın ilk hâli test edilmekte ve öğrenciler filtreleme ile sıralama işlemlerini boş ekran yerine gerçek kayıtlar üzerinde deneyimleyebilmektedir. Bu küçük adım, geliştirme sürecinde doğrulama ve hata ayıklamayı ciddi biçimde kolaylaştırmaktadır.

Yeni Kayıt Ekleme İşlemi
Kullanıcı giriş alanlarına veri girip "Ekle" butonuna bastığında yeni kayıt listeye eklenmektedir. Burada özellikle ad-soyad alanı için boş kontrolü yapılmakta ve kullanıcıya uyarı penceresi gösterilmektedir. Bu yaklaşım, doğrulamanın tüm ayrıntılarını bu aşamada yüklemeden temel kullanıcı hatalarını önlemek için yeterlidir. Yeni kayıt eklendikten sonra giriş alanının temizlenmesi ve odaklanmanın tekrar ilgili alana verilmesi, arayüz akışını daha kullanışlı hâle getirmektedir. Öğrenciler açısından önemli nokta şudur: TableView'e doğrudan "satır çizdirmemekteyiz"; ObservableList'e yeni kayıt eklediğimiz için tablo kendini güncellemektedir. İşte JavaFX veri bağlama mantığının gücü burada görünür hâle gelmektedir.

Arama ile Filtreleme ve Sıralama (Görünüm Listeleri)
Bu ünitenin en öğretici bölümlerinden biri filtreleme ve sıralama kısmıdır. Burada ana veri listesi yerine FilteredList ve SortedList kullanılarak "görünüm listeleri" üretilmektedir. Önce FilteredList ile arama metnine göre hangi kayıtların görünmesi gerektiği belirlenmekte, ardından SortedList ile

kullanıcı sıralama etkileşimi yönetilmektedir. SortedList karşılaştırıcısının TableView karşılaştırıcısına bağlanması sayesinde sütun başlıklarına tıklanınca sıralama doğal biçimde çalışmaktadır.

Burada kavramanız gereken temel ilke şudur: Filtreleme kayıtları silmez; yalnızca kullanıcının o anda gördüğü görünümü değiştirir. Bu yaklaşım hem güvenlidir hem de veri yönetimi açısından daha doğrudur. Ana veri bozulmadığı için kullanıcı arama metnini temizlediğinde tüm kayıtları yeniden görebilmektedir. Sıralama ve filtreleme birlikte kullanıldığında tablo ekranı, öğrencinin gözünde basit bir demo olmaktan çıkıp gerçek bir uygulama davranışı kazanmaktadır.

Seçili Satırı Silme (Başlangıç Düzeyi İçin Buton Tabanlı Yaklaşım)
Ünitede başlangıç düzeyi için satır bazlı ContextMenu yerine buton tabanlı silme yaklaşımı tercih edilmektedir. Bu pedagojik açıdan yerindedir. Çünkü önce SelectionModel üzerinden seçili öğeyi alma mantığı öğrenildiğinde, daha gelişmiş satır etkileşimlerine geçiş çok daha kolay olmaktadır. Seçim değişimini dinleyerek "Seçileni Sil" butonunun etkinliğini güncellemek, arayüzde kullanıcıya doğru zamanda doğru eylemi sunma açısından iyi bir örnektir. Silme işlemi ise seçili kayıt alınarak ana veri listesinden çıkarılmasıyla gerçekleştirilmektedir. Böylece öğrenciler seçim, buton durumu ve veri listesi ilişkisini birlikte görmektedir.

Uygulamayı Çalıştırma ve Beklenen Sonuç
Uygulama tamamlandığında öğrencinin yeni kayıt ekleme, sütunlara göre sıralama, arama ile görünümü daraltma ve seçili satırı silme işlemlerini yapabilmesi beklenmektedir. Bu dört işlev, TableView tabanlı tablo yönetiminin çekirdek iskeletini oluşturmaktadır. Buradaki amaç, tüm ileri düzey özellikleri aynı anda öğretmek değil; TableView'ün veriyle doğru bağlandığı, sütunların doğru gösterildiği ve temel etkileşimlerin güvenli biçimde çalıştığı bir yapı kurmaktır. Bu temeli doğru kurduğunuzda, ilerleyen aşamalarda hücre düzenleme, doğrulama, kalıcılık ve gelişmiş UI davranışlarını çok daha kolay inşa edebileceksiniz.

ÇOK SAYFALI ARAYÜZ KAVRAMININ TEMEL MANTIĞI

Masaüstü uygulamalarda kullanıcı etkileşimi, çoğu zaman tek bir ekranda gerçekleşmez. Kullanıcıdan bilgi alma, bu bilgiyi işleme ve sonucu sunma gibi adımlar genellikle farklı ekranlarda yürütülür. Bu nedenle çok sayfalı arayüz yaklaşımı, modern uygulamaların temel tasarım anlayışlarından biri hâline gelmiştir. Çok sayfalı bir arayüz, bir uygulamanın tek bir pencere içinde, birden fazla ekran sunabilmesini ifade eder.

Bu yaklaşımda amaç, kullanıcıyı gereksiz pencere kalabalığına maruz bırakmadan, adım adım ilerleyen ve sezgisel bir deneyim sunmaktır. Çoklu pencere yaklaşımı yerine tek pencere–çok ekran anlayışının benimsenmesi; kullanılabilirlik, öğrenilebilirlik ve bakım kolaylığı açısından önemli avantajlar sağlar. Kullanıcı, uygulamanın bütünlüğünü kaybetmeden ilerlerken geliştirici açısından da daha kontrol edilebilir bir yapı ortaya çıkar.

JavaFX'te Uygulama Akışı (Application Flow)

Çok sayfalı arayüzlerin temelinde, ekranlar arası geçişlerin belirli bir mantık çerçevesinde kurgulanması yer alır. Bu mantık, uygulama akışı (application flow) olarak adlandırılır. Uygulama akışı; hangi ekrandan sonra hangi ekranın geleceğini, bu geçişlerin hangi koşullarda gerçekleşeceğini ve kullanıcıya nasıl bir yol sunulacağını tanımlar.

JavaFX uygulamalarında akış genellikle iki temel model üzerinden düşünülür: doğrusal ve dallanmış akış. Doğrusal akışta kullanıcı, belirli bir sırayı takip ederek ilerler. Dallanmış akışta ise kullanıcının yaptığı seçimler, birden fazla olası yolun ortaya çıkmasına neden olur. Gerçek hayattaki uygulamaların büyük bir kısmı, bu iki modelin birlikte kullanıldığı karma yapılar içerir.

Akış tasarımının önceden planlanması, uygulamanın hem kullanıcı deneyimini hem de kod yapısını doğrudan etkiler. Plansız geçişler, ilerleyen aşamalarda karmaşık ve sürdürülemez yapılara yol açabilir.

Sahne (Scene) Tabanlı Navigasyon

JavaFX'te ekran geçişlerini yönetmenin temel yollarından biri, sahne (scene) tabanlı navigasyondur. Bu yaklaşımda, tek bir Stage üzerinde farklı Scene nesneleri oluşturulur ve ihtiyaç duyulduğunda sahne değiştirilerek ekran geçişi sağlanır. Scene tabanlı yapı, özellikle küçük ve orta ölçekli uygulamalar için sade ve anlaşılır bir çözüm sunar.

Sahne değişimiyle yapılan navigasyonda her ekran, kendi kök bileşenine sahip bağımsız bir yapı olarak ele alınır. Bu durum, ekranlar arası geçişlerin net biçimde kontrol edilmesini sağlar. Ancak ekran sayısının artmasıyla birlikte sahne yönetimi karmaşıklaşabilir. Bu nedenle sahne tabanlı yaklaşım, uygulamanın ölçeği dikkate alınarak tercih edilmelidir.

Panel (Pane) Tabanlı Navigasyon

Alternatif bir yaklaşım olarak panel tabanlı navigasyon, tek bir scene içinde dinamik içerik değişimini esas alır. Bu yöntemde uygulama, sabit bir çerçeveye sahipken içerik alanı panel değişimiyle güncellenir. BorderPane, StackPane ve benzeri düzen bileşenleri bu yaklaşımın temel yapı taşlarıdır. Panel tabanlı navigasyon, özellikle menü yapısına sahip uygulamalarda etkili bir çözüm sunar. Kullanıcı, ekran değişimi hissetmeden farklı içeriklere ulaşabilir. Ayrıca bu yapı, yeniden kullanılabilir bileşenlerin oluşturulmasını kolaylaştırır. Ancak panel değişimlerinin iyi planlanmaması durumunda kod karmaşıklığı artabilir.

FXML Tabanlı Çok Sayfalı Yapı

JavaFX uygulamalarında çok sayfalı arayüzlerin sürdürülebilir biçimde geliştirilmesinde FXML önemli bir rol oynar. Her ekran için ayrı bir FXML dosyası kullanılması, arayüz tanımı ile iş mantığının ayrılmasını sağlar. Bu ayrım, kodun okunabilirliğini artırır ve bakım sürecini kolaylaştırır. FXML tabanlı yapıda her ekran genellikle kendi Controller sınıfına sahiptir. Controller'ların yalnızca ilgili ekranın sorumluluklarını üstlenmesi, uygulamanın modüler bir yapıya kavuşmasına katkı sağlar. Bu yaklaşım, özellikle büyük ölçekli projelerde tercih edilir.

Ekranlar Arası Veri Akışı

Çok sayfalı bir uygulamada ekran geçişleri çoğu zaman veri aktarımıyla birlikte gerçekleşir. Kullanıcının bir ekranda yaptığı seçimlerin, bir sonraki ekranda anlamlı biçimde kullanılabilmesi gerekir. Bu nedenle ekranlar arası veri akışı, navigasyon tasarımının ayrılmaz bir parçasıdır.

JavaFX uygulamalarında veri aktarımı farklı yöntemlerle gerçekleştirilebilir. Constructor kullanımı, zorunlu verilerin ekran oluşturulurken iletilmesini sağlar. Setter metotları, daha esnek ve duruma bağlı

veri aktarımı için kullanılır. Ortak model sınıfları ise uygulama genelinde kullanılan verilerin merkezi olarak yönetilmesine olanak tanır. Bu yaklaşım, ekranlar arası veri paylaşımını açık ve kontrollü hâle getirir. Hangi yöntemin seçileceği, verinin kapsamına ve kullanım süresine bağlıdır.

Küçük Ölçekli Uygulama Yaklaşımı

Ünitede ele alınan kavramlar, küçük ölçekli bir JavaFX uygulama senaryosu üzerinden bütüncül biçimde değerlendirilebilir. Bu tür bir senaryoda, çok sayfalı arayüz tasarımı, navigasyon ve veri akışı birlikte ele alınır. Amaç, karmaşık yapılar kurmak değil; doğru mimari kararlarla sade ama etkili çözümler üretmektir.

Bu yaklaşım, öğrencinin kendi projelerini geliştirirken yalnızca kod yazmaya değil, uygulamanın genel kurgusunu düşünmeye yönelmesini sağlar.

Genel Sonuç

Bu ünite, JavaFX tabanlı uygulamalarda çok sayfalı arayüzlerin nasıl tasarlanacağına dair temel ve kritik noktaları özetlemektedir. Doğru navigasyon yaklaşımının seçilmesi, akışın planlanması ve veri paylaşımının bilinçli şekilde yönetilmesi hem kullanıcı deneyimi hem de yazılım kalitesi açısından belirleyicidir. Ünitede ele alınan kavramlar, farklı ölçeklerdeki uygulamalara uyarlanabilecek güçlü bir temel sunmaktadır.

ARAYÜZ TANIMLAMA YAKLAŞIMLARI

Bu ünitede kullanıcı arayüzlerinin nasıl tanımlandığı, bu tanımlamanın yazılım geliştirme sürecine olan etkileri ve JavaFX ortamında kullanılan temel yaklaşımlar ele alınmaktadır. Kullanıcı arayüzü, bir uygulamanın kullanıcı ile doğrudan temas kurduğu katman olduğu için yalnızca görsel bir unsur değil; aynı zamanda kullanılabilirlik, sürdürülebilirlik ve geliştirme verimliliği açısından da kritik bir bileşendir. Bu nedenle arayüzün hangi yöntemle oluşturulduğu, uygulamanın genel kalitesini doğrudan etkiler.

JavaFX ortamında arayüz geliştirme süreci temelde iki ana yaklaşım etrafında şekillenir: programatik (kod tabanlı) arayüzler ve deklaratif (FXML tabanlı) arayüzler. Ünitenin temel amacı, bu iki yaklaşımı teknik özellikleriyle tanıtmak, güçlü ve zayıf yönlerini ortaya koymak ve hangi durumlarda hangisinin tercih edilmesi gerektiğine ilişkin bilinçli bir bakış açısı kazandırmaktır.

PROGRAMATİK (KOD TABANLI) ARAYÜZ YAKLAŞIMI

Programatik arayüz yaklaşımında kullanıcı arayüzünü oluşturan tüm bileşenler doğrudan Java kodu içerisinde tanımlanır. Butonlar, metin alanları, etiketler ve düzen bileşenleri nesne olarak oluşturulur; özellikleri ve davranışları yine kod aracılığıyla belirlenir. Bu yaklaşımda arayüz ile uygulama mantığı aynı bağlamda yer alır.

Bu yöntemin en belirgin avantajı, geliştiriciye yüksek düzeyde kontrol ve esneklik sunmasıdır. Arayüz bileşenleri çalışma zamanında koşullara bağlı olarak oluşturulabilir, değiştirilebilir veya kaldırılabilir. Özellikle dinamik yapıya sahip, kullanıcı etkileşimine göre şekil değiştiren arayüzlerde programatik yaklaşım güçlü bir çözüm sunar. Küçük ve basit uygulamalarda hızlı geliştirme imkânı sağlaması da bu yaklaşımı cazip kılar.

Bununla birlikte, uygulama büyüdükçe arayüz kodlarının uygulama mantığı ile iç içe geçmesi, okunabilirliği azaltabilir. Arayüz bileşenlerinin sayısı arttıkça kodun karmaşıklığı artar ve bakım maliyeti yükselir. Bu nedenle programatik yaklaşım, genellikle küçük ölçekli ya da sınırlı arayüz yapısına sahip projelerde daha uygundur.

DEKLARATİF (FXML TABANLI) ARAYÜZ YAKLAŞIMI

Deklaratif arayüz yaklaşımı, kullanıcı arayüzünün nasıl oluşturulacağını değil, nasıl görüneceğini tanımlamaya odaklanır. JavaFX ortamında bu yaklaşım, XML tabanlı FXML dosyaları kullanılarak gerçekleştirilir. FXML dosyaları, arayüz bileşenlerini ve bu bileşenler arasındaki hiyerarşik yapıyı tanımlayan bir yapı sunar.

Bu yaklaşımın temel özelliği, arayüz tanımı ile uygulama mantığının birbirinden ayrılmasıdır. Arayüz FXML dosyasında tanımlanırken, kullanıcı etkileşimleri ve iş mantığı Controller sınıflarında yer alır. Bu ayrım, kodun daha düzenli, okunabilir ve sürdürülebilir olmasını sağlar. Ayrıca ekip çalışmasına dayalı projelerde, tasarımcı ve geliştirici rollerinin ayrılmasına imkân tanır.

FXML tabanlı yaklaşım, özellikle büyük ve uzun vadeli projelerde tercih edilir. Arayüzde yapılacak görsel değişiklikler çoğu zaman uygulama koduna dokunulmadan gerçekleştirilebilir. Ancak çok dinamik arayüz gereksinimlerinde, deklaratif yapı tek başına yeterli olmayabilir ve ek kod desteği gerektirebilir.

YAKLAŞIMLARIN KARŞILAŞTIRILMASI

Programatik ve deklaratif yaklaşımlar, aynı işlevi yerine getirebilseler de yazılım geliştirme süreci üzerinde farklı etkiler oluşturur. Programatik yaklaşım esneklik ve dinamiklik açısından avantaj sağlarken, deklaratif yaklaşım okunabilirlik, bakım kolaylığı ve ekip çalışması açısından öne çıkar. Okunabilirlik açısından bakıldığında, FXML tabanlı arayüzlerde görsel yapı daha net bir şekilde ayrıştırılmıştır. Bakım ve genişletilebilirlik söz konusu olduğunda, deklaratif yaklaşım genellikle daha düşük maliyetlidir. Performans açısından ise iki yaklaşım arasında belirgin bir fark bulunmamakla birlikte, dinamik içerik gerektiren durumlarda programatik yaklaşım daha esnek çözümler sunar.

Bu karşılaştırma, arayüz tanımlama yaklaşımı seçiminin teknik doğruluktan çok bağlama bağlı bir karar olduğunu göstermektedir. Uygulamanın büyüklüğü, geliştirme süresi, ekip yapısı ve gelecekteki genişleme ihtiyacı bu kararı doğrudan etkiler.

HİBRİT YAKLAŞIMLAR

Gerçek dünya JavaFX uygulamalarında en sık karşılaşılan yapı, programatik ve deklaratif yaklaşımların birlikte kullanıldığı hibrit yapılardır. Bu yaklaşımda arayüzün statik yapısı FXML ile tanımlanırken, dinamik ve davranışsal kısımlar Java kodu ile yönetilir. Böylece her iki yaklaşımın

güçlü yönlerinden yararlanılmış olur.
Hibrit yapı, özellikle çok sayfalı arayüzler, panel tabanlı navigasyonlar ve modüler ekran tasarımları için uygundur. Ancak bu yaklaşımın etkili olabilmesi için arayüz yapısının ve veri akışının önceden planlanması gerekir. Aksi hâlde Controller sınıfları gereğinden fazla sorumluluk üstlenebilir ve karmaşıklık artabilir.

UYGULAMA TEMELLİ DEĞERLENDİRME

Ünitenin sonunda, aynı işlevi yerine getiren basit bir JavaFX uygulamasının hem kod tabanlı hem de FXML tabanlı olarak geliştirilmesiyle iki yaklaşım somut olarak karşılaştırılmıştır. Bu uygulama üzerinden yapılan değerlendirme, programatik yaklaşımın hızlı geliştirme avantajını; FXML tabanlı yaklaşımın ise düzenli yapı ve bakım kolaylığı sağladığını açık biçimde ortaya koymaktadır.
Sonuç olarak bu ünite, öğrenciye yalnızca arayüzlerin nasıl oluşturulduğunu değil, hangi bağlamda hangi yaklaşımın tercih edilmesi gerektiğini de öğretmeyi amaçlamaktadır. Arayüz tasarımı, bu bakış açısıyla ele alındığında, rastgele kararların alındığı bir süreç olmaktan çıkarak planlı ve bilinçli bir yazılım tasarım faaliyetine dönüşmektedir.

FXML İLE ARAYÜZ TASARIMI

FXML'in JavaFX İçindeki Konumu

JavaFX uygulamalarında kullanıcı arayüzünün nasıl tanımlandığı, uygulamanın okunabilirliği, sürdürülebilirliği ve ekip içi iş birliği açısından belirleyici bir rol oynar. FXML, JavaFX çatısı altında geliştirilen uygulamalarda arayüz tanımını Java kodundan ayırmayı amaçlayan deklaratif bir yaklaşımdır. XML tabanlı bu yapı, arayüz bileşenlerinin ve yerleşim düzenlerinin betimleyici bir biçimde tanımlanmasına olanak tanır. Bu yaklaşım, arayüz tasarımını yazılımın davranışsal yönlerinden ayrıştırarak daha sade ve yönetilebilir bir yapı sunar.

Kod tabanlı arayüz tanımlamalarında bileşenlerin oluşturulması, yerleştirilmesi ve özelliklerinin ayarlanması doğrudan Java kodu içinde gerçekleştirilir. Bu durum, özellikle arayüz karmaşıklığı arttıkça kodun okunmasını zorlaştırır. FXML ise bu karmaşıklığı azaltmayı ve arayüz tanımını daha görünür hâle getirmeyi hedefler.

FXML Dosya Yapısının Temel Özellikleri

FXML dosyaları XML tabanlı bir sözdizimine sahiptir ve her dosya mutlaka bir kök bileşen ile başlar. Bu kök bileşen, sahnedeki tüm diğer görsel bileşenleri kapsayan ana konteynerdir. VBox, HBox, BorderPane ve AnchorPane gibi yerleşim bileşenleri sıklıkla kök bileşen olarak kullanılmaktadır. Namespace tanımlamaları sayesinde JavaFX bileşenleri FXML dosyası içinde doğru biçimde ifade edilebilir ve FXMLLoader tarafından sorunsuz şekilde işlenir.

FXML yapısında bileşenler hiyerarşik olarak tanımlanır. Bu hiyerarşi, arayüzün görsel düzenini doğrudan yansıtır. Böylece geliştirici, arayüzün yapısını yalnızca FXML dosyasına bakarak kavrayabilir. Bu durum, özellikle büyük ölçekli projelerde ciddi bir avantaj sağlar.

JavaFX Bileşenlerinin FXML ile Tanımlanması

FXML, JavaFX'te yer alan temel kontrol bileşenlerinin tamamını destekler. Button, Label, TextField gibi kontrollerin yanı sıra GridPane, VBox ve HBox gibi yerleşim bileşenleri FXML üzerinden rahatlıkla tanımlanabilir. Bileşenlere ait özelliklerin (property) FXML içinde belirtilmesi, görsel düzenlemelerin Java koduna dokunulmadan yapılabilmesini mümkün kılar.

Bu yaklaşım, arayüz üzerinde sık değişiklik yapılması gereken projelerde zaman kazandırır. Arayüz tasarımına ilişkin kararların koddan bağımsız olarak alınabilmesi, geliştirme sürecini daha esnek hâle getirir.

fx:id Kavramı ve Controller ile İletişim

FXML ile Java kodu arasındaki bağlantı fx:id kavramı üzerinden sağlanır. fx:id, FXML dosyasında tanımlanan bir bileşenin controller sınıfı içerisinden erişilebilir olmasını sağlar. Bu sayede kullanıcıdan alınan girdiler okunabilir, bileşenlerin içerikleri güncellenebilir ve arayüz davranışları kontrol edilebilir.

Controller sınıfında @FXML anotasyonu kullanılarak bu bileşenler Java tarafına bağlanır. fx:id tanımlarının controller sınıfındaki alan adlarıyla birebir uyuşması gerekmektedir. Bu uyum sağlanmadığında, çalışma zamanında hatalar ortaya çıkması kaçınılmazdır.

FXML ve Controller Ayrımı

FXML yaklaşımı, arayüz ile iş mantığının birbirinden ayrılmasını teşvik eder. Controller sınıfı, kullanıcı etkileşimlerine verilen tepkilerin tanımlandığı ve iş mantığının yürütüldüğü katmandır. Bu yapı, Model-View-Controller (MVC) mimari anlayışıyla doğrudan örtüşmektedir.

Arayüz tanımı FXML dosyasında, uygulama davranışı ise controller sınıfında yer aldığında kodun okunabilirliği belirgin biçimde artar. Bu ayrım, özellikle ekip hâlinde geliştirilen projelerde rol dağılımını kolaylaştırır.

FXML Yaşam Döngüsü ve initialize Metodu

FXML dosyalarının yüklenme süreci FXMLLoader tarafından yönetilir. Bu süreçte FXML dosyası okunur, bileşenler oluşturulur, controller sınıfı ile bağlantı kurulur ve son aşamada initialize() metodu çalıştırılır. initialize() metodu, arayüz bileşenlerinin başlangıç ayarlarının yapılması için kritik bir noktadır.

Ancak initialize() metodunun yalnızca başlangıç konfigürasyonları için kullanılması gerekir. Ağır iş mantığının bu metot içine yerleştirilmesi, uygulamanın bakımını zorlaştırabilir.

FXML ile Olay Yönetimi

FXML, olay yönetiminin de deklaratif olarak tanımlanmasına olanak tanır. Örneğin bir butonun

tıklanma olayı, FXML dosyasında doğrudan controller sınıfındaki bir metoda bağlanabilir. Bu yöntem, arayüz ile davranış arasındaki ilişkiyi açık ve takip edilebilir hâle getirir.
Olay yönetiminin bu şekilde yapılandırılması, arayüz tanımının sade kalmasını sağlarken, davranışsal kodun controller içinde merkezi olarak yönetilmesine imkân tanır. Bu durum, uygulamanın genişletilebilirliğini artırır.

FXML Kullanımına Genel Bir Bakış

FXML, özellikle orta ve büyük ölçekli JavaFX projelerinde tercih edilen bir arayüz tanımlama yaklaşımıdır. Okunabilirlik, bakım kolaylığı ve ekip çalışmasına uygunluk gibi avantajlar sunar. Bununla birlikte küçük ve tek geliştiricili projelerde kod tabanlı arayüzler hâlâ geçerli bir seçenek olmaya devam etmektedir. Bu nedenle yaklaşım seçimi, proje ölçeği ve geliştirme bağlamı dikkate alınarak yapılmalıdır.

Grafiksel Kullanıcı Arayüzü (GUI) Kavramı

Grafiksel Kullanıcı Arayüzü (Graphical User Interface – GUI), kullanıcı ile yazılım sistemi arasında görsel bileşenler aracılığıyla etkileşim kurulmasını sağlayan yapıdır. Geleneksel komut satırı (CLI) uygulamalarında kullanıcı metin tabanlı komutlar girerek işlem yaparken, GUI tabanlı uygulamalarda butonlar, metin alanları, listeler, menüler ve görsel simgeler aracılığıyla etkileşim gerçekleşir. Bu yaklaşım, kullanıcı deneyimini iyileştirmekte ve yazılımların daha erişilebilir hâle gelmesini sağlamaktadır.

GUI tasarımında temel amaç, işlevselliği estetik ve düzenli bir yapı içerisinde sunmaktır. Bu nedenle arayüz geliştirme süreci yalnızca teknik bir faaliyet değil, aynı zamanda kullanıcı deneyimi (UX) ve kullanıcı arayüzü (UI) tasarım prensiplerini de kapsayan disiplinler arası bir çalışmadır. Renk seçimi, hizalama, boşluk kullanımı ve görsel denge gibi unsurlar yazılımın kalitesini doğrudan etkiler.

JavaFX, Java programlama dili kullanılarak modern GUI uygulamaları geliştirmek için tasarlanmış bir kütüphanedir. JavaFX ile masaüstü uygulamaları geliştirilebilir; aynı zamanda multimedya, animasyon ve grafik bileşenleri de desteklenir. JavaFX'in temel yapısı Stage (pencere), Scene (içerik alanı) ve Node (bileşen) kavramlarına dayanır. Stage uygulamanın ana penceresini temsil ederken, Scene pencere içindeki içerik alanını ifade eder. Node ise sahne grafiğinde yer alan tüm arayüz elemanlarını kapsar.

JavaFX'te Mimari Yaklaşım: MVC Yapısı

JavaFX uygulamalarında genellikle Model–View–Controller (MVC) mimarisi tercih edilir. Bu yapı, yazılımın daha düzenli ve sürdürülebilir olmasını sağlar. MVC yaklaşımında:

Model, uygulamanın veri yapısını ve iş kurallarını içerir.

View, kullanıcı arayüzünü temsil eder.

Controller, kullanıcı etkileşimlerini yakalar ve iş mantığını yönetir.

Bu ayrım sayesinde arayüz ile iş mantığı birbirinden ayrılır. Böylece tasarım değişiklikleri yapılırken iş mantığı etkilenmez ve algoritmik değişiklikler arayüz yapısını bozmaz. Özellikle büyük projelerde bu ayrım yazılımın bakımını kolaylaştırır.

JavaFX ortamında View genellikle FXML dosyası ile tanımlanır. Controller sınıfı kullanıcı olaylarını yönetir. Model ise veri işlemlerini içerir. Bu yapı, yazılım geliştirme sürecinde modülerliği artırır.

FXML Dosya Yapısı

FXML, JavaFX arayüzlerini XML tabanlı olarak tanımlamaya yarayan bir işaretleme dilidir. FXML dosyasında arayüz bileşenleri hiyerarşik bir yapı içerisinde tanımlanır. Örneğin bir VBox içerisine Label ve TextField eklenmesi, XML etiketleri aracılığıyla ifade edilir.

FXML kullanmanın temel avantajı, tasarım ile programlama kodunun ayrılmasıdır. Arayüz yapısı FXML dosyasında bulunurken, olayların ve iş mantığının kodu Controller sınıfında yer alır. Bu durum, tasarımcı ile yazılımcının ayrı çalışabilmesine olanak tanır.

FXML dosyasında bir bileşenin Controller sınıfında kullanılabilmesi için fx:id tanımlaması yapılır. Ayrıca buton gibi etkileşimli bileşenlerde onAction özelliği kullanılarak ilgili metod adı belirtilir. Böylece FXML ile Controller arasında bağlantı kurulur.

SceneBuilder Ortamı ve Temel Özellikleri

SceneBuilder, FXML dosyalarını görsel olarak tasarlamaya yarayan bir araçtır. Sürükle–bırak yöntemi sayesinde kullanıcı, arayüz bileşenlerini kod yazmadan tasarım alanına ekleyebilir. Yapılan her değişiklik arka planda FXML dosyasına otomatik olarak yansıtılır.

SceneBuilder arayüzü dört temel bölümden oluşur:

Library (Bileşen Kütüphanesi): Layout yapıları ve kontrol bileşenleri burada yer alır.

Design Area (Tasarım Alanı): Uygulamanın görsel önizlemesinin yapıldığı bölümdür.

Hierarchy Paneli: Bileşenlerin sahne grafiğindeki hiyerarşik yapısını gösterir.

Inspector Paneli: Seçilen bileşenin özelliklerini düzenlemeye yarar.

Inspector paneli üç sekmeden oluşur: Properties (özellikler), Layout (yerleşim ayarları) ve Code (fx:id ve olay tanımlama). Bu yapı hem görsel hem teknik düzenlemeyi mümkün kılmaktadır.

Layout (Yerleşim) Yapıları

Arayüz tasarımında düzenin sağlanabilmesi için layout yapıları kullanılır. Layout'lar bileşenlerin konumlandırılmasını ve hizalanmasını belirler. VBox, bileşenleri dikey olarak sıralar. Giriş ekranlarında veya form tasarımlarında sıklıkla tercih edilir. Alignment ve spacing ayarları ile

hizalama ve boşluk kontrol edilir. HBox, bileşenleri yatay olarak yerleştirir. Örneğin bir Label ve TextField'ın yan yana gösterilmesi için kullanılır. AnchorPane, bileşenleri belirli kenarlara sabitlemeye imkân tanır. Pencere boyutu değiştiğinde bileşenlerin konumunun korunmasını sağlar. GridPane, satır ve sütun mantığında düzen sağlar. Tablo biçiminde veri giriş formları için uygundur. Bu yapı, düzenli ve hizalı formlar oluşturmak açısından avantajlıdır. Doğru layout seçimi, arayüzün düzenli ve kullanıcı dostu olmasını doğrudan etkiler.

Temel Bileşenler

JavaFX'te en sık kullanılan bileşenler arasında; Label (metin göstermek için kullanılır), TextField (tek satırlık veri girişi sağlar), PasswordField (girilen veriyi gizler), Button (kullanıcı etkileşimini başlatır) ve ComboBox (açılır liste sunar) olduğu söylenebilir. Bu bileşenler SceneBuilder'da sürükle-bırak yöntemiyle eklenir ve Inspector panelinden özellikleri düzenlenir.

Özellik (Property) Düzenleme ve Olay Tanımlama

Bileşenlerin metin, renk, hizalama, yazı tipi gibi özellikleri Inspector panelinden ayarlanır. Layout sekmesinde konum ve sabitleme ayarları yapılabilir. Code sekmesinde fx:id ve onAction tanımlamaları gerçekleştirilir.

Bir butona onAction özelliği tanımlandığında, ilgili metod Controller sınıfında yazılır. Böylece tasarım ile işlevsellik bağlantılı hâle gelir. Bu süreç MVC yapısının uygulanmasını sağlar.

Hızlı Prototipleme Yaklaşımı

Hızlı prototipleme, yazılımın erken aşamada basitleştirilmiş bir modelinin oluşturulmasıdır. Bu yaklaşımın amacı tasarımın görselleştirilmesi, kullanıcı geri bildiriminin alınması ve hataların erken tespit edilmesidir. SceneBuilder, hızlı prototipleme için oldukça uygundur. Öğrenciler birkaç dakika içinde bir giriş ekranı veya hesaplama arayüzü tasarlayabilir. Bu sayede tasarım doğrulaması yapılır ve geliştirme süreci hızlanır.

NetBeans ile Entegrasyon ve Çalıştırma Süreci

SceneBuilder'da tasarlanan FXML dosyası, NetBeans ortamında JavaFX projesi içinde kullanılır. Uygulama Application sınıfındaki main metodu aracılığıyla başlatılır. Controller sınıfı doğrudan çalıştırılmaz. Bu yüzden ana sınıf üzerinden yüklenir.

Bu süreçte Controller class tanımlamasının doğru yapılması, fx:id eşleşmelerinin birebir olması ve onAction metodunun yazılması önemlidir. Aksi hâlde çalışma zamanı hataları oluşabilir.

Genel Değerlendirme

JavaFX ve SceneBuilder birlikte kullanıldığında, görsel tasarım ile programlama süreçleri dengeli biçimde yürütülebilir. Öğrenciler önce tasarım mantığını öğrenir, ardından işlevselliği kod ile tamamlar. Bu yaklaşım, bilişsel yükü azaltır ve öğrenmeyi kolaylaştırır.

MVC yapısına uygun, modüler ve sürdürülebilir uygulamalar geliştirmek mümkün hâle gelir. Layout yapılarının doğru kullanımı, property düzenleme becerisi ve olay bağlantılarının doğru kurulması etkili bir JavaFX uygulamasının temelini oluşturur. Sonuç olarak, SceneBuilder destekli JavaFX eğitimi hem görsel tasarım becerisi hem de yazılım geliştirme yetkinliği kazandıran bütüncül bir öğretim yaklaşımı sunmaktadır.

FORM TABANLI UYGULAMALARA GİRİŞ

Form tabanlı uygulamalar, kullanıcıdan veri almak ve bu verileri işlemek amacıyla geliştirilen yazılım yapılarıdır. Günümüzde kullanılan masaüstü uygulamalarının büyük bir kısmı, kullanıcı ile sistem arasındaki etkileşimi formlar aracılığıyla sağlar. Bir giriş ekranı, kayıt paneli, bilgi güncelleme sayfası ya da hesaplama arayüzü aslında birer form uygulamasıdır.

Formlar, belirli alanlar aracılığıyla kullanıcıdan bilgi toplayan ve bu bilgileri sistem içinde işleyen arayüz yapılardır. Bu alanlar genellikle metin kutuları, seçim listeleri, işaretleme kutuları ve butonlardan oluşur. Kullanıcı bu alanlara veri girer ve belirli bir işlemi başlatır. Böylece kullanıcı ile yazılım arasında düzenli ve kontrollü bir veri alışverişi gerçekleşir.

Formların yazılım geliştirme sürecindeki önemi oldukça büyüktür. Çünkü bir yazılımın doğruluğu ve güvenilirliği büyük ölçüde kullanıcıdan alınan verilerin doğruluğuna bağlıdır. Yanlış veya eksik veri ile çalışan bir sistem, hatalı sonuçlar üretir. Bu nedenle form yapısı yalnızca görsel bir arayüz değil, aynı zamanda veri kontrol mekanizmasının ilk aşamasıdır.

Günlük Hayatta Form Örnekleri ve Dijital Dönüşüm

Form kavramı yalnızca yazılım alanına özgü değildir. Günlük hayatta sıklıkla karşılaşılan başvuru belgeleri, kayıt formları, banka evrakları ve resmi başvuru dokümanları form örnekleridir. Bu formların ortak amacı, belirli bilgileri eksiksiz ve düzenli bir şekilde toplamaktır.

Dijital dönüşüm ile birlikte kâğıt formlar yerini elektronik formlara bırakmıştır. Elektronik formlar sayesinde girilen veriler anında kaydedilebilir, analiz edilebilir ve farklı sistemlere aktarılabilir. Ayrıca hatalı girişler anında tespit edilerek kullanıcı uyarılabilir. Bu durum, dijital form sistemlerinin daha güvenli ve verimli olmasını sağlamaktadır.

Dolayısıyla yazılım geliştiricilerin form tasarımını doğru planlaması hem veri kalitesi hem de kullanıcı deneyimi açısından büyük önem taşımaktadır.

Yazılımda Form Kullanımının Amaçları

Yazılım uygulamalarında formlar genellikle üç temel amaç için kullanılır: veri toplama, veri güncelleme ve veri görüntüleme. Veri toplama aşamasında kullanıcıdan yeni bilgiler alınır. Örneğin bir öğrenci kayıt sistemi, öğrencinin adını, numarasını ve bölümünü toplar.

Veri güncelleme sürecinde mevcut kayıtlar düzenlenir. Kullanıcı daha önce girdiği bilgileri değiştirebilir. Veri görüntüleme ise sistemde bulunan bilgilerin kullanıcıya düzenli biçimde sunulmasıdır. Bu üç süreç de form yapısı üzerinden gerçekleştirilir. Dolayısıyla form, yazılım ile kullanıcı arasındaki temel iletişim köprüsüdür.

JAVAFX'TE TEMEL FORM BİLEŞENLERİ

Arayüz Bileşenlerinin Rolü

JavaFX, masaüstü uygulama geliştirmek için kullanılan modern bir arayüz kütüphanesidir. JavaFX'te form tasarlamak için çeşitli bileşenler kullanılır. Bu bileşenler kullanıcıdan veri almak veya kullanıcıyı bilgilendirmek amacıyla tasarlanmıştır. Form bileşenleri doğru seçildiğinde hem veri giriş hataları azalır hem de kullanıcı deneyimi iyileşir.

Label ve Bilgilendirme Yapıları

Label bileşeni, kullanıcıya bilgi vermek için kullanılır. Bu bileşen veri almaz, yalnızca metin gösterir. Bir formda yer alan alanların açıklanması için gereklidir. Örneğin "Ad Soyad", "E-posta" veya "Şifre" gibi başlıklar Label ile oluşturulur. Label kullanımı, formun anlaşılabilirliğini artırır ve kullanıcıyı yönlendirir.

Veri Giriş Bileşenleri

TextField, kullanıcıdan tek satırlık metin almak için kullanılır. Bu alan genellikle isim, e-posta veya kullanıcı adı gibi bilgiler için tercih edilir. PasswordField ise şifre girişinde kullanılır ve girilen karakterleri gizler. Bu durum güvenlik açısından önemlidir. ComboBox, kullanıcının hazır seçenekler arasından seçim yapmasını sağlar. Böylece yazım hataları engellenir. DatePicker, tarih girişini standartlaştırır ve format hatalarının önüne geçer. RadioButton ve CheckBox ise seçim işlemleri için kullanılır. RadioButton tek seçim, CheckBox ise çoklu seçim sağlar.

BUTON VE OLAY YÖNETİMİ

Event Kavramı

Event, kullanıcının uygulama üzerinde gerçekleştirdiği bir eylemdir. Butona tıklamak, klavyeden tuşa basmak veya seçim yapmak birer olaydır. Olay yönetimi, bu eylemler gerçekleştiğinde sistemin nasıl

tepki vereceğini belirler.

Buton Tıklama Mekanizması

Butonlar form uygulamalarının aktif bileşenleridir. setOnAction yöntemi ile butona bir olay tanımlanır. Butona basıldığında belirlenen kod bloğu çalışır. Bu mekanizma sayesinde formdan alınan veriler işlenir. Lambda ifadeleri, olay tanımlamayı daha sade ve anlaşılır hale getirir. Bu yöntem özellikle basit işlemler için idealdir.

GİRDİ DOĞRULAMA (INPUT VALIDATION)

Doğrulamanın Tanımı ve Önemi

Girdi doğrulama, kullanıcıdan alınan verilerin belirlenen kurallara uygun olup olmadığını kontrol etme işlemidir. Bu işlem yapılmazsa program hatalı çalışabilir veya yanlış veriler kaydedilebilir. Doğrulama, yazılım güvenliğinin temel adımlarından biridir.

Temel Doğrulama Türleri

Boş alan kontrolü, en temel doğrulama türüdür. Sayısal veri kontrolü, sayı olması gereken alanlara harf girilmesini engeller. Uzunluk kontrolü, belirli karakter sınırlarını denetler. Şifre kontrolü, güvenli giriş sağlamak için kullanılır.

JAVAFX'TE DOĞRULAMA TEKNİKLERİ

If-else ile Koşul Kontrolü

If-else yapısı, temel doğrulama işlemlerinde kullanılır. Alan boş mu? Uzunluk yeterli mi? gibi sorular bu yapı ile cevaplanır.

Try-catch ile Hata Yönetimi

Try-catch yapısı, hata oluşturabilecek işlemleri güvenli biçimde yönetir. Böylece program çökmez, hata kontrol altına alınır.

Regex ile Format Kontrolü

Regex, belirli bir metin desenini kontrol etmek için kullanılır. Özellikle e-posta doğrulamasında yaygındır.

Görsel Geri Bildirim

Alanların kırmızı veya yeşil renkle işaretlenmesi, kullanıcıya görsel geri bildirim sağlar. Bu yöntem kullanıcı deneyimini artırır.

Alert ile Mesaj Gösterme

Alert sınıfı ile kullanıcıya hata veya bilgi mesajı gösterilir. Bu yapı, kullanıcıyı doğru giriş yapmaya yönlendirir.

FORM VERİLERİNİ İŞLEME VE MODELLEME

Formdan alınan veriler değişkenlere aktarılır ve program içinde kullanılabilir hale gelir. Basit model sınıfları oluşturularak veriler nesne haline getirilebilir. Bu yaklaşım yazılımın daha düzenli ve sürdürülebilir olmasını sağlar.

GENEL DEĞERLENDİRME VE SONUÇ

Form tabanlı uygulamalar, masaüstü yazılım geliştirmenin temel yapı taşlarından biridir. JavaFX ile geliştirilen uygulamalarda form bileşenleri, layout yapıları, olay yönetimi ve doğrulama teknikleri birlikte kullanılır. Doğru yapılandırılmış bir form uygulaması; güvenli, kullanıcı dostu ve hataya dayanıklı bir yazılım ortaya çıkarır. Bu beceriler, daha ileri düzey yazılım geliştirme çalışmalarının temelini oluşturur.

Dosya tabanlı veri işleme, programların çalışma süresi dışında da veriyi kalıcı olarak saklayabilmesini sağlayan temel bir yazılım yaklaşımıdır. Bellek üzerinde tutulan değişkenler uygulama sonlandığında kaybolurken, dosyalar aracılığıyla saklanan veriler sistem üzerinde kalıcılığını korur. Bu nedenle dosya işlemleri, veri yönetiminin en temel bileşenlerinden biridir. Özellikle küçük ve orta ölçekli masaüstü uygulamalarda, yapılandırma bilgileri, kullanıcı kayıtları, log dosyaları ve rapor çıktıları gibi verilerin metin dosyalarında saklanması oldukça yaygın bir uygulamadır.

Metin dosyaları, karakter tabanlı veri saklama biçimidir ve satır sonu karakterleri ile ayrılmış metinsel içerikten oluşur. Bu yapı, hem insan tarafından okunabilir olması hem de platformlar arası taşınabilirlik sağlaması açısından önemli avantajlar sunar. Ancak ham verinin yalnızca dosyada bulunması yeterli değildir; verinin doğru biçimde okunması, işlenmesi ve gerektiğinde güncellenmesi gerekir. Bu noktada dosya tabanlı veri işleme, yalnızca veri depolama değil, aynı zamanda veri erişimi, dönüştürme ve hata yönetimi süreçlerini de kapsayan bütüncül bir yapı sunar.

Java programlama dilinde dosya işlemleri, temel olarak java.io ve java.nio.file paketleri aracılığıyla gerçekleştirilir. File sınıfı, dosya ve dizinlerin temsil edilmesini sağlar; dosya oluşturma, silme, varlığını kontrol etme ve listeleme gibi işlemleri destekler. Ancak gerçek veri aktarımı, akış (stream) yapıları aracılığıyla yapılır. Java I/O mimarisi, veri ile program arasındaki iletişimi akış kavramı üzerinden tanımlar. Akışlar iki temel kategoriye ayrılır: bayt (byte) akışları ve karakter (character) akışları. Metin dosyaları ile çalışırken karakter akışları tercih edilir, çünkü bu akışlar veriyi Unicode karakterler biçiminde işler ve metinsel içerik için uygun bir altyapı sunar.

Metin dosyasına veri yazma işlemi belirli bir algoritmik yapıya dayanır. Öncelikle dosyaya bağlı bir çıkış akışı oluşturulur, ardından veri bu akış üzerinden dosyaya aktarılır ve işlem tamamlandığında akış kapatılır. Bu üç adımlı yapı, kaynak yönetimi açısından kritik öneme sahiptir. Akışın kapatılmaması durumunda dosya kilitlenebilir veya veri kaybı oluşabilir. Benzer şekilde, dosyadan veri okuma süreci de giriş akışı oluşturma, veriyi döngü içinde okuma ve akışı kapatma adımlarından oluşur. BufferedReader sınıfının readLine() yöntemi, satır satır veri okuma imkânı sağlayarak metin dosyalarının etkin biçimde işlenmesine olanak tanır.

Dosya tabanlı veri işlemede hata yönetimi ayrı bir önem taşır. Dosyanın bulunamaması, erişim izinlerinin yetersiz olması ya da veri formatının beklenenden farklı olması gibi durumlar çalışma zamanında istisna (exception) üretir. Bu nedenle IOException ve FileNotFoundException gibi hataların yakalanması ve kullanıcıya anlamlı mesajlarla bildirilmesi gerekir. Modern Java sürümlerinde kullanılan try-with-resources yapısı, akışların otomatik olarak kapatılmasını sağlayarak kaynak sızıntılarını önleyen güvenli bir programlama yaklaşımı sunar.

Java NIO (New Input/Output) yapısı, dosya işlemlerini daha sade ve esnek bir biçimde gerçekleştirmeyi mümkün kılar. Path ve Files sınıfları sayesinde dosya içeriği tek satırda okunabilir veya yazılabilir. Bu yaklaşım, özellikle küçük veri kümelerinde kodun daha okunabilir ve kısa olmasını sağlar. Bununla birlikte büyük veri dosyalarında satır bazlı okuma ve tamponlama teknikleri performans açısından daha avantajlıdır.

Bölüm kapsamında ayrıca JavaFX ortamında dosya tabanlı uygulama geliştirme ele alınmıştır. JavaFX, masaüstü uygulamaları için modern bir grafiksel kullanıcı arayüzü altyapısı sunar. TextArea, TextField, Button ve FileChooser gibi bileşenler kullanılarak basit bir metin dosyası düzenleyici uygulaması geliştirilebilir. Bu uygulama, kullanıcının bir dosya seçmesini, içeriğini görüntülemesini, düzenlemesini ve tekrar kaydetmesini sağlar. Böylece dosya tabanlı veri işleme yalnızca arka planda çalışan bir mekanizma olmaktan çıkar, kullanıcı ile doğrudan etkileşim kuran bir sistem bileşeni hâline gelir.

Karakter kodlaması konusu da metin dosyalarıyla çalışırken göz ardı edilmemelidir. Farklı sistemlerde

varsayılan kodlama değişebileceğinden, UTF-8 gibi standart bir kodlamanın açıkça belirtilmesi veri bütünlüğü açısından önemlidir. Aksi hâlde Türkçe karakterler gibi özel semboller yanlış görüntülenebilir.

Sonuç olarak dosya tabanlı veri işleme, kalıcı veri yönetiminin temel yapı taşlarından biridir. Java'nın sunduğu I/O ve NIO altyapıları sayesinde metin dosyalarına veri yazmak, veriyi satır satır okumak ve dinamik olarak güncellemek mümkündür. JavaFX ile bu işlemler kullanıcı dostu arayüzlerle desteklenebilir. Bu bölüm, metin dosyalarıyla veri saklama ve okuma süreçlerini hem kavramsal hem de uygulamalı boyutuyla ele alarak, öğrencilerin kalıcı veri yönetimi konusunda sağlam bir teknik temel oluşturmasını amaçlamaktadır.

DERS ADI Görsel Programlama II
ÜNİTE ADI VERİ TABANI DESTEKLİ JAVAFX UYGULAMALARI
ÜNİTE NO 10
YAZAR Dr.Öğr.Üyesi Önder YILDIRIM

Bu bölümde, modern masaüstü yazılım geliştirme sürecinin temel bileşenlerinden biri olan veritabanı destekli uygulama geliştirme, JavaFX ve JDBC teknolojileri kullanılarak kapsamlı ve uygulamalı bir biçimde ele alınmıştır. Günümüzde bankacılık sistemlerinden e-ticaret platformlarına, üniversite otomasyonlarından hastane bilgi sistemlerine kadar pek çok yazılımın veri tabanı ile çalıştığı düşünüldüğünde, bir yazılım geliştiricinin hem kullanıcı arayüzü tasarlayabilmesi hem de veritabanı işlemlerini etkin şekilde yönetebilmesi kritik bir beceri olarak öne çıkmaktadır. Bu bölüm, öğrencilerin daha önce öğrendikleri dosya tabanlı veri saklama yöntemlerinden profesyonel veritabanı yönetimine geçiş yapmalarını sağlayacak şekilde yapılandırılmıştır.

Bölümün ilk kısmında JavaFX mimarisi tanıtılmış ve Stage, Scene, Scene Graph ve Node kavramları üzerinden bir JavaFX uygulamasının temel bileşenleri açıklanmıştır. Bu yapı sayesinde arayüz bileşenlerinin hiyerarşik olarak düzenlenmesi, yönetilmesi ve dinamik olarak değiştirilmesi mümkün olmaktadır. JavaFX'in sunduğu Button, TextField, ComboBox, TableView gibi kontrol bileşenleri ile VBox, HBox, GridPane gibi düzen yöneticileri kullanılarak modern ve kullanıcı dostu arayüzler oluşturulabileceği gösterilmiştir. FXML teknolojisi ile arayüz tasarımının uygulama mantığından ayrılması sağlanmış; NetBeans IDE ve Scene Builder entegrasyonu sayesinde öğrencilerin sürükle-bırak yöntemiyle arayüz tasarlamaları mümkün hale getirilmiştir. Bu yaklaşım, yazılım geliştirme sürecinde kodun daha düzenli, okunabilir ve sürdürülebilir olmasını desteklemektedir.

İkinci bölümde JDBC (Java Database Connectivity) API'si tanıtılmış ve Java uygulamalarının MySQL, PostgreSQL, SQLite veya SQL Server gibi ilişkisel veritabanlarına nasıl bağlanabileceği adım adım açıklanmıştır. JDBC sürücülerinin türleri, bağlantı URL'leri ve DriverManager kullanımı incelenmiş; try-with-resources yapısı ile bağlantıların güvenli şekilde kapatılması vurgulanmıştır. SQL komutlarının Statement ve PreparedStatement sınıfları ile yürütülmesi gösterilmiş ve PreparedStatement kullanımının SQL enjeksiyon saldırılarına karşı sağladığı güvenlik avantajları açıklanmıştır. Bu süreçte CRUD işlemleri (Create, Read, Update, Delete) bir Data Access Object (DAO) sınıfı aracılığıyla gerçekleştirilmiş ve uygulamanın katmanlı mimari ile tasarlanması sağlanmıştır.

Model–View–Controller (MVC) mimarisi bölümün önemli konularından biri olmuştur. Bu yaklaşımda Model veri ve iş kurallarını yönetirken, View kullanıcı arayüzünü temsil eder ve Controller kullanıcı etkileşimlerini yönetir. JavaFX projelerinde FXML dosyalarının View katmanını, Controller sınıflarının kontrol mantığını ve DAO sınıflarının Model katmanını temsil ettiği açıklanmıştır. Bu yapı, yazılımın bakımını kolaylaştırmakta, test edilebilirliğini artırmakta ve büyük projelerde kod karmaşasını önlemektedir.

Veritabanı verilerinin görselleştirilmesi için JavaFX'in TableView bileşeni kullanılmış ve ObservableList yapısı ile veri bağlama mekanizmaları tanıtılmıştır. Bu sayede modeldeki değişikliklerin kullanıcı arayüzüne otomatik yansıması sağlanmıştır. Ayrıca kullanıcı girdilerinin doğrulanması, hata yönetimi ve try-catch yapıları ile güvenli veri işleme konuları ele alınmıştır. Uzun süren veritabanı işlemlerinde arayüzün donmasını önlemek için JavaFX Task ve Service sınıfları kullanılarak concurrency kavramı tanıtılmıştır.

Bölümün ilerleyen kısımlarında JavaFX uygulamalarının görsel açıdan geliştirilmesi için CSS ile stilleme yöntemleri gösterilmiş; butonların, tablo başlıklarının ve hata mesajlarının özelleştirilmesi uygulamalı olarak anlatılmıştır. Daha büyük ölçekli projeler için Hibernate veya JPA gibi Nesne–İlişkisel Haritalama (ORM) araçlarının kullanımı tanıtılarak JDBC kodlarının nasıl sadeleştirilebileceği açıklanmıştır. ORM yaklaşımı sayesinde geliştiricilerin SQL sorguları yerine nesne tabanlı veri işlemleri gerçekleştirebileceği vurgulanmıştır.

Bölüm sonunda geliştirilen öğrenci kayıt sistemi örneği, JavaFX ve JDBC kullanarak tam işlevli bir masaüstü uygulamasının nasıl oluşturulacağını göstermektedir. Bu örnek; arayüz tasarımı, veritabanı bağlantısı, veri listeleme, güncelleme ve silme işlemleri ile birlikte modern yazılım geliştirme

süreçlerinin temelini oluşturmaktadır. Bu bölümde kazanılan beceriler, öğrencilerin hem akademik projelerinde hem de gerçek hayat yazılım geliştirme süreçlerinde kullanabilecekleri güçlü bir altyapı sunmakta; ileri düzey mobil, web veya yapay zekâ tabanlı projeler için sağlam bir temel oluşturmaktadır.

VERİ GÖRSELLEŞTİRME

Veri görselleştirme, sayısal ve kategorik veri kümelerinin grafiksel temsiller aracılığıyla anlamlandırılmasını, yorumlanmasını ve sunulmasını sağlayan sistematik bir yaklaşımdır. Günümüzde dijitalleşmenin hız kazanmasıyla birlikte veri üretimi de benzeri görülmemiş bir biçimde artmaktadır. Ancak verinin niceliksel olarak artması, tek başına anlamlı bilgi üretildiği anlamına gelmez. Ham veriler, uygun yöntemlerle işlenmediği, düzenlenmediği ve hedef kitleye uygun biçimde sunulmadığı sürece karar alma süreçlerine katkı sağlayamaz. İşte tam bu noktada veri görselleştirme, ham veri ile anlamlı bilgi arasındaki köprüyü kuran temel bir araç olarak karşımıza çıkmaktadır.
Tablo biçiminde sunulan sayısal değerler, çoğu zaman veri kümesi içindeki eğilimlerin, dönemsel dalgalanmaların, kategoriler arası farklılıkların ve değişkenler arası ilişkilerin fark edilmesini güçleştirir. Oysa insan zihni, görsel örüntüleri ve uzamsal ilişkileri sayısal listelere ya da metinsel ifadelere kıyasla çok daha hızlı ve etkili biçimde işlemektedir. Grafikler, bu bilişsel avantajı kullanarak verideki artış ve azalışları, dağılımları, kümelenmeleri ve aykırı değerleri anlık olarak görünür hâle getirir. Bu yönüyle görselleştirme, yalnızca bir sunum tekniği değil, aynı zamanda bilişsel süreçleri doğrudan destekleyen analitik bir araçtır.
Etkili bir veri görselleştirme süreci, teknik bilgi kadar analitik düşünme becerisi de gerektirir. Geliştirici ya da tasarımcı, öncelikle elindeki verinin yapısını çözümlemeli, değişken türlerini (kategorik, sayısal, zamansal) doğru biçimde tanımlamalı ve bu veri yapısına en uygun grafik türünü belirlemelidir. Bu süreçte temel amaç, karmaşık ve çok boyutlu bilgiyi sadeleştirerek kullanıcıya açık, anlaşılır ve yönlendirici bir görsel sunum hazırlamaktır. Doğru grafik türü seçimi ve uygun tasarım ilkelerinin uygulanması, verinin doğru yorumlanmasını sağlarken, yanlış seçimler verinin mesajını zayıflatabilir ya da izleyiciyi tamamen yanlış yönlendirebilir.

JAVAFX GRAFİK ALTYAPISI

JavaFX, masaüstü uygulamaları geliştirmek için kapsamlı bir kullanıcı arayüzü kütüphanesi sunmakta ve veri görselleştirme ihtiyaçları için özel olarak tasarlanmış zengin bir grafik bileşenleri seti içermektedir. Bu bileşenler, JavaFX'in sahne grafiği mimarisi içinde yer alır ve tüm arayüz elemanlarının ortak atası olan Node sınıfından türetilir. Bu mimari yapı sayesinde grafikler, butonlar, etiketler, tablolar ve diğer arayüz bileşenleri ile bütünleşik biçimde çalışabilir, aynı sahne üzerinde bir arada yer alabilir ve ortak düzenleme kurallarına tabi olabilir.
JavaFX grafik mimarisinin merkezinde Chart sınıfı yer almaktadır. Bu soyut sınıf, tüm grafik türleri için ortak olan başlık, açıklama, arka plan gibi temel öğeleri tanımlar. İki eksenli grafikler ise XYChart soyut sınıfından türetilmiştir. LineChart, BarChart, AreaChart, ScatterChart ve BubbleChart bu soyut sınıfın alt sınıflarıdır. Bu hiyerarşik yapı, farklı grafik türlerinin ortak bir altyapıyı paylaşmasını sağlarken, her bir grafik türünün kendine özgü çizim davranışını da korumasına olanak tanır. Geliştirici, yalnızca eksen türlerini ve veri serilerini tanımlar; veri noktalarının koordinat sistemine yerleştirilmesi, çizgilerin ya da çubukların çizilmesi gibi karmaşık işlemler sistem tarafından otomatik olarak gerçekleştirilir.
JavaFX grafiklerinin en önemli teknik özelliklerinden biri, otomatik aralık ayarlama yeteneğidir. Grafik oluşturulurken eksenlerin minimum ve maksimum değerleri, eklenen veri serilerine göre otomatik olarak hesaplanır. Bu özellik, geliştiriciyi manuel ölçek ayarlama zahmetinden kurtarır ve dinamik veri setleriyle çalışmayı kolaylaştırır. Ayrıca JavaFX grafikleri, yerleşik animasyon desteğine sahiptir. Veri noktaları eklendiğinde ya da güncellendiğinde, bu değişiklikler görsel geçiş efektleriyle yansıtılabilir. Bu durum, özellikle canlı veri akışının izlendiği uygulamalarda kullanıcı deneyimini olumlu yönde etkileyen önemli bir faktördür.

JAVAFX GRAFİK TÜRLERİ

Veri yapısı ile grafik türü arasında doğrudan ve belirleyici bir ilişki vardır. Her grafik türü, belirli bir veri yapısını ve analiz amacını destekleyecek biçimde tasarlanmıştır. Bu nedenle grafik türlerinin güçlü ve zayıf yönlerini, hangi bağlamlarda etkili olduklarını bilmek, doğru görselleştirme kararları alabilmek için zorunludur.

Çizgi Grafiği (LineChart)

Zaman serisi verileri için en sık tercih edilen grafik türüdür. Veri noktalarını kronolojik sırayla birleştiren çizgiler, süreklilik gösteren değişimleri ve eğilimleri belirgin biçimde ortaya koyar. Öğrenci başarı oranlarının dönemsel değişimi, bir ürünün aylık satış performansı ya da hava sıcaklığının günlük seyri gibi sürekli değişkenler çizgi grafikle etkili biçimde görselleştirilebilir.

Sütun Grafiği (BarChart)
Kategorik verilerin karşılaştırılmasında kullanılan temel araçtır. Her bir kategori için çizilen dikdörtgen çubukların uzunluğu, o kategoriye ait sayısal değeri temsil eder. Farklı bölümlere ait öğrenci sayıları, ürün kategorilerine göre satış miktarları ya da şehirler bazında nüfus yoğunlukları sütun grafikle net biçimde karşılaştırılabilir.

Pasta Grafiği (PieChart)
Bir bütünün parçalara ayrıştırılarak oransal dağılımın gösterilmesi gerektiğinde tercih edilir. Dairesel dilimlerin merkez açıları, ilgili kategorinin toplam içindeki payını yansıtır. Ancak kategori sayısı arttıkça dilimlerin karşılaştırılması zorlaşır ve grafiğin okunabilirliği hızla düşer. Bu nedenle pasta grafikler, sınırlı sayıda kategori için uygun bir seçimdir.

Dağılım Grafiği (ScatterChart)
İki sayısal değişken arasındaki ilişkiyi analiz etmek için kullanılır. Her veri noktası, X ve Y koordinatlarına karşılık gelen bir sembolle işaretlenir. Noktaların dağılım deseni, değişkenler arasında pozitif ya da negatif bir korelasyon olup olmadığını, verinin kümelenme eğilimini ve aykırı değerleri ortaya çıkarır.

Alan Grafiği (AreaChart)
Çizgi grafiğin altında kalan alanın doldurulmasıyla elde edilir. Birden fazla serinin birikimli toplamını ya da zaman içindeki kümülatif değişimi göstermek için kullanılır. Bu grafik türü, toplam miktar ve bileşen katkılarını aynı anda görselleştirmek isteyen durumlar için idealdir.
Grafik türü seçimi yapılırken yalnızca veri yapısı değil, aynı zamanda hedef kitlenin özellikleri, sunum ortamı ve iletilmek istenen ana mesaj da dikkate alınmalıdır. Karmaşık bir veri kümesini en doğru grafikle sunmak, analiz sürecinin verimliliğini artırdığı gibi, karar vericilerin doğru sonuçlara ulaşmasını da kolaylaştırır.

EKSENLER VE VERİ SERİLERİ
Grafiklerin temel yapı taşlarını eksenler ve veri serileri oluşturur. Eksenler, verinin grafik alanında nasıl konumlandırılacağını belirleyen referans çizgileridir. JavaFX'te kategorik veriler için CategoryAxis, sayısal veriler için ise NumberAxis sınıfı kullanılır. Eksenlerin doğru tanımlanması, grafiğin okunabilirliğini doğrudan etkileyen en önemli faktörlerden biridir.
Veri serisi (Series), grafik üzerinde birlikte temsil edilen ve ortak bir mantıksal kümeye ait olan veri noktalarının bütününü ifade eder. Her seri, bir isim ile tanımlanır ve bu isim grafik lejantında görüntülenerek izleyicinin seriyi tanımlamasını sağlar. Aynı grafik üzerinde birden fazla seri gösterilebilir. Bu özellik, örneğin farklı yıllara ait satış verilerini ya da farklı bölgelerin performans göstergelerini aynı grafik üzerinde karşılaştırmalı olarak analiz etme imkânı sunar.
Veri noktaları XYChart.Data nesneleri aracılığıyla oluşturulur. Her bir Data nesnesi, bir X ve bir Y değeri içerir. Bu nesneler, serilerin veri listelerine eklenerek grafiğe aktarılır. Bu programatik yapı, grafiklerin esnek biçimde yönetilmesine olanak tanır. Geliştirici, uygulama çalışma zamanında yeni veri noktaları ekleyebilir, mevcut veri noktalarını güncelleyebilir ya da silebilir. Bu özellik, gerçek zamanlı veri akışına sahip sistemlerde grafiklerin dinamik olarak güncellenmesine olanak sağlar.

CSS'IN TEMEL BILEŞENLERI

Bu bölümün ana ekseni JavaFX uygulamalarındaki arayüz tasarımının CSS aracılığıyla nasıl kontrol edilebileceğini göstermektir. CSS, web tasarımından bilinen basamaklı stil sayfaları yaklaşımının JavaFX bileşenlerine uyarlanmış halidir ve içerik ile görsel katmanı birbirinden ayırarak hem görsel zenginlik hem de bakım kolaylığı sağlar. JavaFX CSS kuralları üç temel bileşene dayanır: seçici (selector), özellik (property) ve değer (value). Seçici, hangi bileşenin veya bileşen grubunun stil kuralından etkileneceğini belirler. Örneğin, .button seçicisi, uygulamada kullanılan tüm butonları hedef alır. Özellik, bileşenin hangi görsel niteliğinin (renk, yazı tipi, kenarlık, arka plan gibi) değiştirileceğini tanımlar. JavaFX'de bu özellikler -fx- önekiyle başlar; bu önek, web CSS'inden ayırt edilmesini sağlar. Değer ise, söz konusu özelliğin nasıl uygulanacağını belirtir (örneğin -fx-font-size: 15px, -fx-text-fill: white). Bu bileşenlerin bir araya gelişiyle, basit bir CSS kuralı bile düğmelerin yazı tipinden renklerine kadar birçok özelliği değiştirebilir.

CSS'İN JAVAFX UYGULAMALARINA EKLENMESİ

Harici Stil Dosyaları

Bu yöntem, CSS kurallarını .css uzantılı dosyalara yazarak projede tutmayı içerir. Dosya içindeki stil kurallarını sınıf veya kimlik seçicileriyle tanımlar, ardından bu dosyayı Java kodu üzerinden sahneye (Scene) veya kapsayıcıya (Parent) ekleriz. Scene.getStylesheets().add(...) çağrısı, dosyanın tüm sahnedeki bileşenlere uygulanmasını sağlar. Birden fazla dosya eklendiğinde, listede son eklenen dosya önceki dosyalardaki aynı özellikleri geçersiz kılar. Bu yöntem, tasarımın koddan ayrılmasını sağlar, böylece tasarımcı ve geliştirici iş birliği kolaylaşır.

Parent (Ebeveyn) Düzeyinde Stil Dosyaları

Daha dar kapsamda bir stil uygulamak istiyorsak, CSS dosyasını belirli bir kapsayıcıya ekleriz. Örneğin, bir kenar çubuğu için sadece VBox içine eklenen sidebar-styles.css dosyası, sahnenin geri kalanını etkilemeden sadece bu bölgenin stilini belirler. Çakışma durumlarında, kapsayıcıya eklenen stil dosyası sahne düzeyinde eklenenden daha öncelikli kabul edilir. Bu yöntem, ekranın farklı bölgelerinde farklı tema kullanmak veya modüler bir tasarım oluşturmak için idealdir.

Inline (Bileşen Düzeyi) Stil Kullanımı

Node.setStyle(String css) metodu ile bir bileşene doğrudan CSS özelliği tanımlanabilir. Inline stil, en yüksek önceliğe sahiptir. Yani aynı özelliği tanımlayan harici veya kapsayıcı stilleri geçersiz kılar. Bununla birlikte, inline stile başvurmanın dezavantajları büyüktür. Kodun içinde görünüş tanımlarını sakladığı için hem tekrar kullanımı zordur hem tasarım ile işlevsel kodu birbirine karıştırır. Bu nedenle, inline stil sadece küçük denemeler veya dinamik olarak değişmesi gereken çok özel durumlar için önerilir.

Bir JavaFX projesinde stil sisteminin öncelik sıralaması "kaskat" mantığıyla işler. En yüksek öncelik inline stillerdedir. Bunu kapsayıcıya eklenen stiller, sahneye eklenen stiller ve en düşük olarak Modena gibi varsayılan tema dosyası takip eder. Uygulamanın genel tutarlılığını korurken istisna durumları yönetmek için bu hiyerarşiyi anlamak önemlidir.

STİL SEÇİCİLERİ VE ÖZELLEŞTİRME YÖNTEMLERİ

Sınıf Seçicileri (Class Selectors)

.button, .label gibi varsayılan sınıf isimleri, JavaFX bileşenlerinin çoğu için mevcuttur. Bunları kullanarak tüm butonların yazı tipini, kenarlığını veya arka plan rengini tek bir yerden değiştirebilirsiniz. Geliştiriciler ayrıca getStyleClass().add("ozel-sinif") ile bileşenlere kendi sınıflarını ekleyebilir ve CSS dosyalarında bu özel sınıflar için kurallar tanımlayarak aynı türden bileşenler arasında farklılık oluşturabilir. Böylece butonların bir kısmı "birincil", diğerleri "ikincil" olarak tanımlanabilir ve farklı görünebilir.

ID Seçicileri

#myButton gibi id seçicileri, tek bir bileşeni hedeflemek için kullanılır. Node.setId(String id) metodu ile bileşen kimliği atanır ve CSS dosyasında #myButton { ... } kuralı ile stil verilir. Aynı ID'nin birden

fazla bileşen tarafından kullanılması doğru değildir. Bu nedenle ID seçicileri bileşen bazında spesifik özelleştirmeler için uygundur.

Tip Seçicileri (Type Selectors)
Bileşen sınıfının adını doğrudan kullanarak (Button, Label) tüm o sınıftan türetilmiş bileşenleri hedefler. Ancak varsayılan stil sınıfları çoğu zaman aynı işlevi sağladığı ve daha esnek kombinasyonlara izin verdiği için pratikte daha az tercih edilir. Örneğin, .label sınıfı ile Label tip seçicisi aynı elementi hedefler ancak sınıf seçicisi birden fazla sınıfa eklenebilir.

PSEUDO-SINIFLAR VE DİNAMİK STİL DEĞİŞİMİ

Arayüz tasarımının sadece statik görünümlerden ibaret değildir. Kullanıcı etkileşimlerine bağlı dinamik geri bildirimler ile zenginleştirilebilir. Pseudo-sınıflar, bileşenlerin belirli anlık durumlarını ifade eden CSS eklentileridir ve iki nokta (:) ile tanımlanırlar.

hover
Fare imleci bileşenin üzerindeyken etkin olur. Örneğin, butonun üzerine gelindiğinde arka plan rengini değiştirebilirsiniz.

focused
Bileşen klavye veya fare ile seçildiğinde uygulanır. Metin alanına odaklandığınızda kenarlığın belirginleşmesini sağlayabilirsiniz.

disabled
Bileşen devre dışı bırakıldığında (disable olduğunda) uygulanır. Bu durumda bileşenin renkleri soluk hale gelebilir.

pressed
Özellikle butonlarda, basılı tutulduğu anda tetiklenir. Buton basılı iken gölgeli bir görünüm verilebilir.

selected
RadioButton veya CheckBox gibi seçilebilir bileşenler seçildiğinde uygulanır.

Özel Pseudo Sınıflar
Java tarafında, PseudoClass nesnesi ve pseudoClassStateChanged(PseudoClass, boolean) metodu ile bileşene yeni bir durum eklenebilir. Örneğin, bir ShinyLabel sınıfı tanımlanıp, "shiny" adında bir özel pseudo-sınıf kazandırılabilir. Bu durum BooleanProperty aracılığıyla kontrol edilir ve değeri değiştiğinde CSS tarafındaki :shiny kuralı otomatik olarak etkinleşir. Bu yöntem, Java kodu ile CSS'i temiz bir şekilde ayırarak bakım ve genişletilebilirliği artırır.
Pseudo-sınıflar sayesinde, kullanıcı deneyimi zenginleştirilir ve arayüz daha sezgisel hale gelir. Basit bir buton için bile, normal, üzerine gelindiğinde, basılı tutulduğunda ve devre dışı kaldığındaki görünümleri tanımlayarak profesyonel ve kullanıcı dostu uygulamalar geliştirebilirsiniz.

MODENA TEMASI VE TEMA ÖZELLEŞTİRME

JavaFX 8 ile gelen Modena teması, her uygulamanın varsayılan stilini belirler. Butonlardan kaydırma çubuklarına kadar tüm bileşenler bu tema ile uyumlu görünür. Bu sayede, hiçbir CSS yazılmasa dahi uygulama modern ve tutarlı bir görünüme sahiptir. Bununla birlikte, kurumsal kimlik veya kişisel ihtiyaçlar gereği temanın özelleştirilmesi gerekebilir.

Temayı Kopyalayıp Düzenlemek
modena.css dosyası projeye kopyalanarak üzerinde değişiklik yapılır. Renk paleti, yazı tipleri, kenarlıklar ve diğer stiller değiştirilerek tamamen farklı bir tema oluşturulabilir. Bu yöntem ile uygulamanın tüm bileşenlerinde bütüncül bir değişiklik yapılır.

Kısmi Özelleştirme
Sadece değiştirilmek istenen bileşenlerin veya özelliklerin tanımlandığı yeni bir CSS dosyası oluşturulur ve sahneye eklenir. JavaFX'in basamaklı stil mantığı sayesinde, bu dosyada tanımlanmayan diğer özellikler Modena temasından gelmeye devam eder. Bu yöntem, daha hafif bir yapı sunar ve bakım sırasında tüm temayı yeniden düzenlemek yerine sadece gerekli bölümlere müdahale edilir.
Yeni temanın sahneye eklenmesi için, CSS dosyasının getResource() ile bulunup toExternalForm() ile URL formatına dönüştürüldüğü ve scene.getStylesheets().add() metodu ile sahneye bağlanır. Böylece, sahnedeki tüm bileşenler bu temanın kurallarına göre stilize edilir.

Kişisel Bütçe Takip Uygulaması Geliştirme Süreci
Bu ünitede, JavaFX çatısı kullanılarak küçük ölçekli bir masaüstü uygulaması olan Kişisel Bütçe Takip Uygulaması geliştirilmiştir. Uygulama, önceki on iki ünitede ele alınan konuların bütünleşik bir proje içinde uygulanmasını sağlamıştır. Proje geliştirme süreci sekiz ana aşamadan oluşmuş olup her aşamada ilgili kavramlar, teknik gerekçeler ve sık karşılaşılan hatalar ele alınmıştır.
Proje Planlama ve Yapısının Oluşturulması
Birinci aşamada proje planlama ve gereksinim analizi yapılmıştır. Uygulamanın veri girişi, veri listeleme, veri görselleştirme ve veri kalıcılığı olmak üzere dört temel özelliği belirlenmiştir. Projenin genel mimarisi, yazılım mühendisliğinde yaygın olarak kullanılan katmanlı mimari yaklaşımına göre tasarlanmıştır: sunum katmanı (FXML ve CSS), iş mantığı katmanı (PrimaryController) ve veri erişim katmanı (DatabaseHelper). Kullanılacak teknolojiler ve her birinin seçim gerekçeleri tablo halinde sunulmuştur.

İkinci aşamada proje yapısı oluşturulmuştur. NetBeans IDE üzerinde Maven tabanlı FXML JavaFX Maven Archetype kullanılarak BudgetTracker adlı proje oluşturulmuştur. Proje yapılandırma dosyası olan pom.xml dosyasına SQLite JDBC bağımlılığı eklenmiş ve module-info.java dosyasında requires, opens ve exports ifadelerinin ne anlama geldiği açıklanmıştır.
Veritabanı Tasarımı ve Bağlantısı
Üçüncü aşamada veritabanı tasarımı gerçekleştirilmiştir. İşlem kayıtlarını saklamak için id, tarih, tur, kategori, aciklama ve tutar alanlarından oluşan islemler tablosu tasarlanmıştır. Her alanın veri tipi ve kısıtlamaları gerekçeleriyle birlikte açıklanmıştır. DatabaseHelper yardımcı sınıfı, tek sorumluluk ilkesine uygun olarak veritabanı işlemlerini tek bir sınıfta toplamıştır. try-with-resources yapısı ile otomatik kaynak yönetimi sağlanmış, PreparedStatement ile SQL enjeksiyonuna karşı güvenli sorgu yazımı ele alınmıştır. Islem model sınıfı ile veritabanı satırlarının Java nesnelerine dönüştürülmesi sağlanmış, getter metot adlandırma kurallarının JavaFX PropertyValueFactory mekanizmasıyla olan ilişkisi açıklanmıştır.
FXML ile Arayüz Tasarımı
Dördüncü aşamada FXML ile bildirimsel arayüz tasarımı yapılmıştır. BorderPane ana kapsayıcı olarak kullanılmış ve pencereyi üst, alt, sol, sağ ve orta olmak üzere beş bölgeye ayırma mantığı açıklanmıştır. TabPane bileşeni ile İşlemler ve Grafikler olmak üzere iki sekmeli bir yapı oluşturulmuştur. İşlemler sekmesinde DatePicker, ComboBox, TextField ve Button bileşenlerinden oluşan veri giriş formu ile beş sütunlu TableView bileşeni; Grafikler sekmesinde PieChart ve BarChart bileşenleri yerleştirilmiştir. fx:id ile Java kodu arasındaki bağlantı mekanizması ve onAction ile olay işleme bağlantısı açıklanmıştır.
Denetleyici Sınıfı ve İş Mantığı
Beşinci aşamada iş mantığı katmanı olan PrimaryController sınıfı geliştirilmiştir. Initializable arayüzü ve initialize metodunun yapıcı metottan farkı, bileşenlerin bağlanma zamanlaması açısından açıklanmıştır. ObservableList kavramı ve TableView ile otomatik senkronizasyon mekanizması ele alınmıştır. ComboBox bileşenlerinde lambda ifadeleri ile dinamik içerik güncelleme, PropertyValueFactory ile tablo sütun bağlantısı, Alert sınıfı ile kullanıcı bildirimleri ve hata yönetimi uygulamalı olarak ele alınmıştır. App.java dosyasında FXMLLoader, Scene ve Stage kavramları açıklanmıştır.
Grafik Görselleştirme
Altıncı aşamada grafik görselleştirme eklenmiştir. Sekme değişim dinleyicisi mekanizması ile grafiklerin yalnızca görüntülendiğinde hesaplanması sağlanmıştır. PieChart ile gider kategorilerinin oransal dağılımı ele alınmıştır: LinkedHashMap ile kategori bazlı toplama işlemi ve merge metodunun çalışma mantığı açıklanmıştır. BarChart ile toplam gelir-gider karşılaştırması oluşturulurken XYChart.Series ve XYChart.Data kavramları ile veri noktalarının grafiğe eklenmesi ele alınmıştır.
CSS ile Arayüz Özelleştirme ve Test
Yedinci aşamada CSS ile arayüz özelleştirilmiştir. JavaFX CSS ile web CSS arasındaki temel fark olan -fx- öneki açıklanmıştır. CSS seçici türleri, hover efekti ile görsel geri bildirim ve style.css dosyasının sahneye bağlanma yöntemi uygulamalı olarak ele alınmıştır.
Son aşamada uygulama test edilmiştir. Yedi farklı fonksiyonel test senaryosu tanımlanmış ve

uygulanmıştır. Kullanıcı deneyimi testlerinde pencere boyutlandırma davranışı, kategori tutarlılığı ve grafik doğruluğu değerlendirilmiştir. Geliştirme sürecinde sık karşılaşılan hata türleri, nedenleri ve çözümleri tablo halinde sunulmuştur. Projenin katmanlı mimari yapısı özetlenerek gerçek dünya projelerinde de kullanılan bu yapının avantajları vurgulanmıştır.

JAVAFX İLE GÜNCEL VE SÜRDÜRÜLEBİLİR UYGULAMALAR

Masaüstü Uygulama Geliştirmenin Günümüzdeki Yeri

Masaüstü uygulamalar, dijital ekosistemde yerini korumaya devam eden güçlü yazılım çözümleridir. Web ve mobil teknolojilerin yaygınlaşmasına rağmen; özellikle kurumsal sistemler, veri yoğun uygulamalar ve kapalı ağ ortamları masaüstü çözümler için hâlâ önemli bir kullanım alanı oluşturmaktadır. Yerel kaynaklara doğrudan erişim, yüksek performans gereksinimleri ve güvenlik ihtiyaçları masaüstü yazılımları vazgeçilmez kılmaktadır.

JavaFX, bu bağlamda modern arayüz geliştirme ihtiyaçlarına cevap veren, platform bağımsız ve nesne tabanlı bir yapı sunar. Sahne grafiği (scene graph) mimarisi sayesinde arayüz bileşenleri sistematik biçimde yönetilebilir ve görsel–mantıksal ayrım net biçimde sağlanabilir. Bu yapı, hem küçük ölçekli projelerde hem de daha karmaşık uygulamalarda sürdürülebilir bir geliştirme ortamı sunar.

Güncel Kullanım Senaryoları

JavaFX; eğitim yazılımlarında, iç sistem uygulamalarında ve veri odaklı masaüstü çözümlerde etkin şekilde kullanılabilir. Özellikle form tabanlı veri giriş ekranları, raporlama arayüzleri ve grafiksel veri gösterimi gerektiren uygulamalar için uygun bir altyapı sağlar.

Kurumsal ortamlarda geliştirilen personel takip sistemleri, stok yönetim yazılımları, muhasebe uygulamaları ve analiz araçları gibi sistemlerde arayüz performansı ve kararlılık kritik öneme sahiptir. JavaFX'in güçlü kontrol bileşenleri ve CSS desteği bu tür uygulamalarda hem işlevsel hem de görsel açıdan dengeli çözümler üretmeye imkân tanır.

Ölçeklenebilirlik ve Kod Organizasyonu

Sürdürülebilir bir yazılım geliştirme süreci, kodun yalnızca çalışmasıyla değil; okunabilir, genişletilebilir ve bakım yapılabilir olmasıyla ölçülür. Küçük bir uygulama zamanla yeni modüller eklenerek büyüyebilir. Bu noktada paket yapısının doğru tasarlanması, iş mantığının arayüzden ayrılması ve sınıflar arası bağımlılıkların minimize edilmesi büyük önem taşır.

Model–View–Controller (MVC) yaklaşımı, arayüz ve iş mantığının ayrıştırılmasını destekler. FXML kullanımı, bu ayrımı güçlendirerek arayüz tasarımını koddan bağımsız hâle getirir. Bu yaklaşım, özellikle ekip çalışmalarında kod çakışmalarını azaltır ve bakım sürecini kolaylaştırır.

Arayüz Tasarımında İyi Uygulamalar

Başarılı bir masaüstü uygulama yalnızca teknik olarak doğru çalışmakla kalmaz; kullanıcı açısından anlaşılır ve rahat kullanılabilir olmalıdır. Tutarlılık, sadelik ve görsel hiyerarşi arayüz tasarımının temel ilkelerindendir.

Bileşenlerin yerleşiminde gereksiz karmaşıklıktan kaçınılması, kullanıcıyı yormayan renk ve yazı tipi seçimleri yapılması ve mantıksal gruplaşmaların açık olması kullanıcı deneyimini doğrudan etkiler. CSS kullanımı sayesinde tema yönetimi sağlanabilir ve kurumsal kimliğe uygun arayüzler geliştirilebilir.

Performans ve Kaynak Kullanımı

Masaüstü uygulamalarda performans yalnızca işlem hızını değil, kullanıcı deneyimini de belirler. Büyük veri tabloları, yoğun grafik işlemleri veya arka planda çalışan uzun süreçler arayüzün donmasına neden olabilir. Bu nedenle zaman alıcı işlemlerin arka planda yürütülmesi ve kullanıcıya geri bildirim verilmesi gerekir.

JavaFX'te gözlemlenebilir özellikler (properties) ve dinleyiciler (listeners) kullanılarak dinamik arayüz davranışları oluşturulabilir. Ancak gereksiz dinleyici kullanımının bellek tüketimini artırabileceği unutulmamalıdır. İhtiyaç duyulmayan dinleyicilerin kaldırılması performans açısından önemlidir.

Bakım ve Güncelleme Süreci

Uzun ömürlü yazılım geliştirme anlayışı, kodun zamanla değişeceği gerçeğini kabul eder. Yeni özelliklerin eklenmesi, hataların giderilmesi veya arayüzde yapılacak revizyonlar için kodun esnek bir yapıda olması gerekir. Anlamlı sınıf isimleri, açıklayıcı metot adları ve düzenli paket organizasyonu

bakım sürecini kolaylaştırır.
Modüler yapı, bağımlılıkların azaltılması ve arayüz–mantık ayrımı sürdürülebilirliğin temel taşlarıdır. Bu yaklaşım sayesinde uygulama büyüdükçe karmaşıklık kontrol altında tutulabilir.

JavaFX'in Diğer Teknolojilerle Entegrasyonu
Modern masaüstü uygulamalar genellikle tek başına çalışmaz. Veritabanları, web servisleri ve REST tabanlı API'ler ile etkileşim kurar. JavaFX uygulamaları JDBC aracılığıyla veritabanlarına bağlanabilir; HTTP istemcileri üzerinden web servisleriyle iletişim kurabilir.
Bu entegrasyon, masaüstü uygulamaların yalnızca yerel sistemlerde değil, dağıtık mimariler içinde de kullanılabilmesini sağlar. Böylece hibrit çözümler geliştirmek mümkün hâle gelir.

Öğrenci Açısından Kazanımlar
JavaFX ile arayüz geliştirme süreci; nesne tabanlı düşünme becerisini güçlendirir, olay güdümlü programlama mantığını öğretir ve kullanıcı deneyimi perspektifi kazandırır. Arayüz–iş mantığı ayrımı, yazılım mühendisliği prensiplerinin erken aşamada kavranmasını sağlar.
Bu beceriler yalnızca JavaFX ile sınırlı kalmaz; web, mobil ve diğer masaüstü teknolojilerine geçişte de temel oluşturur.

Gelecek Perspektifi
Yazılım teknolojileri sürekli değişmektedir. Ancak arayüz tasarım ilkeleri, modüler yapı ve sürdürülebilir kod anlayışı kalıcıdır. JavaFX, açık kaynaklı yapısı ve genişletilebilir mimarisi sayesinde gelişmeye devam etmektedir.
Alternatif masaüstü teknolojileri bulunsa da, arayüz geliştirme prensiplerini kavramış bir geliştirici için platform değişimi teknik bir adaptasyon sürecine dönüşür. Asıl kalıcı olan, yazılımı sürdürülebilir biçimde tasarlama becerisidir.