Bu rapor kapsamında, Edge Impulse ile eğitilen makine öğrenmesi modelinin ESP32-CAM donanımı üzerinde çalıştırıldığı TinyML tabanlı bir nesne tanıma sistemi geliştirdiğimiz projemizde kullanılan yazılım mimarilerinin gerçekleştirimine ve görev bazlı uygun tasarım desenlerinin seçimine yer verilmiştir.

Sistemimiz; veri toplama, model eğitimi, modelin gömülü cihaza dağıtımı (deployment) ve gerçek zamanlı test gibi ardışık adımları içermektedir. Bu adımlar doğrultusunda yazılım mimarisinde daha önce belirlenen Katmanlı Mimari (Layered Architecture) ile Bileşen Tabanlı Mimari (Component-Based Architecture) yapıları temel alınmıştır.

Bu mimariler sayesinde sistemimiz sunum (girdi/çıktı), iş mantığı (model eğitimi ve sınıflandırma), veri erişimi (kamera ve sensör), uygulama (ESP32 kontrol akışı) gibi alt yapılara ayrılmış, bileşenlerin birbirinden bağımsız çalışması sağlanmıştır. Bu yapı, proje içerisinde uygulanan tasarım desenleri ile yapısal açıdan örtüşmektedir.

Tasarım desenleri (design patterns), yazılım geliştirme sürecinde karşılaşılan sorunlara tekrar kullanılabilir çözümler sunan tasarımsal yaklaşımlardır. Bu projede de sistemin modülerliğini, yeniden kullanılabilirliğini ve esnekliğini artırmak amacıyla uygun desenler belirlenmiştir. Bunlar: Observer Patterns (Gözlemci Deseni), State Patterns (Durum Deseni), Strategy Pattern (Strateji Deseni) ve Singleton Pattern'dir. Her biri belirli görevlerin gerçekleştiriminde referans olarak kullanılmıştır.

Yazının ilerleyen bölümlerinde, projemizde tercih edilen tasarım desenleri; her biri için "amaç" (intent), "kapsam ve ihtiyaç" (motivation), "uygulanabilirlik" (applicability), "yapısal gösterim" (structure) ve "proje bağlamında gerçekleştirim" (implementation) başlıkları altında detaylandırılmıştır. Ayrıca her desenin proje mimarisiyle olan ilişkisi structure başlıkları altında bir blok diyagram aracılığıyla görsel olarak sunulmuştur.

Projemizde kullanılan ESP32-CAM gibi gömülü sistemlerde bellek ve işlem gücü kısıtlı olduğundan, tasarım desenleri klasik nesne yönelimli mimarilerde olduğu gibi sınıf ve fonksiyonlarla tam olarak uygulanmamıştır. Ancak sistemin davranışı ve modüler yapısı, Observer, State ve Strategy gibi desenlerin mantıksal karşılıklarını yansıtmaktadır. Bu nedenle ilgili desenler, uygulama seviyesinde soyutlama yerine işlevsel ayrım olarak projeye entegre edilmiştir.

1. Observer Pattern (Gözlemci Deseni)

1) Intent

Bir nesnede değişiklik olduğunda, bu değişiklikten haberdar olması gereken diğer nesnelerin otomatik olarak bilgilendirilmesini sağlar.

2) Motivation

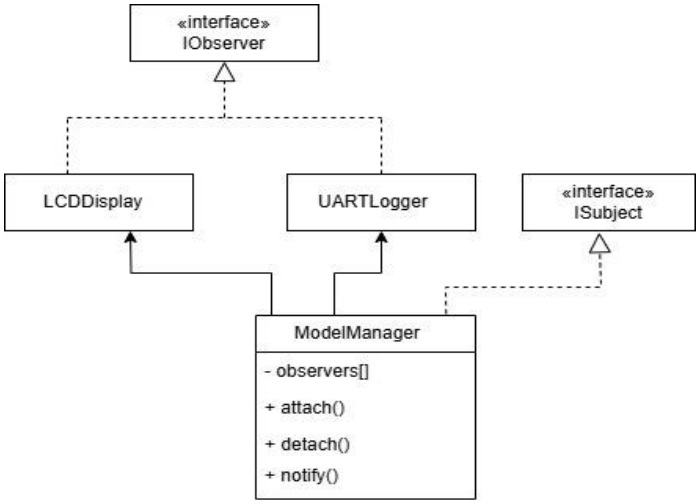
Projemizde, ESP32-CAM üzerinde çalışan TinyML modeli bir nesne algıladığında farklı bileşenlerin (örneğin LCD ekran, UART çıkışı, GPIO pini) buna tepki vermesi gerekir. Bu tetiklemelerin merkezi kontrolle yapılması karmaşık ve esnek olmayan bir yapıya yol açar. Observer pattern ile bu durum loosely-coupled hale getirilir.

3) Applicability

- Gözlemlenen bir nesnede (subject) durum değişiklikleri varsa,
- Bu değişikliklere tepki vermesi gereken birden fazla bağımlı nesne varsa.
  Bu desen, olay tabanlı gömülü sistemlerde oldukça yaygındır.

4) Structure

→ Model çıktısını paylaşır, gözlemciler uygun tepkiyi verir.



5) Implementation

- ESP32 üzerinde model çıktısını alan fonksiyon bir “subject” görevi görür. LCD yazdırma, UART ile gönderme gibi işlemler “observer” bileşenlerdir. Her biri, yalnızca modele abone olur ve çıktıya göre işlem yapar.
- Kodda doğrudan Observer arayüzü ve attach() / notify() fonksiyonları tanımlı olmasa da, model çıktılarına göre birden fazla bileşenin (örneğin LCD ekran, UART çıkışı, GPIO tetikleme) eş zamanlı olarak tepki vermesi sağlanmaktadır. Bu yapı, Observer deseninin temel amacına yani bir olay gerçekleştiğinde bağlı nesneleri bilgilendirmeye uygun şekilde kurgulanmıştır. Her bileşen model çıktısını doğrudan değil, model sonuçlarını yöneten merkezi yapı üzerinden almaktadır. Böylece gevşek bağlı (loosely-coupled) bir yapı elde edilmiştir.

2. State Pattern (Durum Deseni)

1) Intent

Bir nesnenin iç durumu değiştiğinde, davranışını da değiştirmesini sağlar. Nesne, farklı davranışları temsil eden durum nesneleri arasında geçiş yapar.

2)Motivation

Sistemimizde üç ana durum vardır:

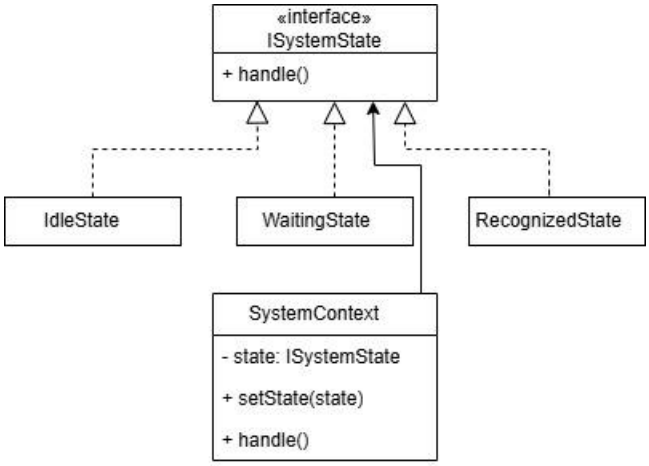
- Model yüklenmedi
- Model yüklendi ama nesne algılanmadı
- Nesne algılandı

Bu durumlara göre sistem farklı şeyler yapmalıdır. Kodun if-else yığınına boğulmaması için bu desen idealdir.

3) Applicability

- Sistemde açıkça tanımlı birden fazla durum varsa,
- Her durumda sistem farklı davranış sergiliyorsa,
- Durum geçişleri sıklıkla oluyorsa.

4) Structure



5) Implementation

- SystemContext adında bir nesne, içinde bulunduğu duruma göre davranır. Her durum (örn. WaitingState, RecognizedState, IdleState) ayrı bir sınıf olarak tanımlanır ve ortak bir arayüzü (SystemState) uygular.
- Sistemimizde cihazın çalışması üç temel duruma ayrılmıştır: “Beklemede”, “Model çalıştırılıyor”, “Nesne algılandı”. Bu durumlara göre sistemin davranışı değişmektedir. Kod içinde Waiting, Processing, Detected gibi özel sınıflar kullanılmasa da, her durum için farklı iş akışları ve tepki mekanizmaları tanımlanmıştır. Durum değişiklikleri genellikle if blokları ile kontrol edilse de sistemin genel akışı State Pattern mantığına uygun şekilde yönetilmiştir. Bu durumlar kodun farklı bölümlerinde davranışsal ayrışmaya neden olur; dolayısıyla desenin ruhu korunmuştur.

3. Strategy Pattern (Strateji Deseni)

1) Intent

Bir işlemi gerçekleştirme yöntemini (stratejiyi) dışarıdan belirlenebilir hale getirerek, sistemin esnekliğini artırır.

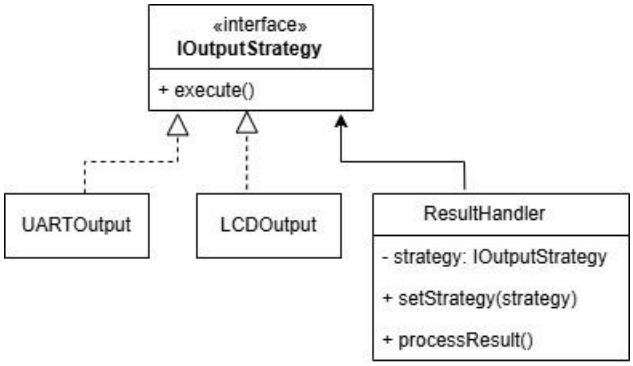
2) Motivation

Projemizde farklı nesneler (elma, havuç, biber) algılandığında farklı işlem stratejileri uygulanmak istenebilir. Örneğin biri UART'tan gönderilirken diğeri LCD’ye yazdırılabilir.

3) Applicability

- Farklı senaryolarda farklı davranışlar gerekiyor ama ortak bir mantık çatısı korunmak isteniyorsa
- Stratejiler runtime'da (çalışma anında) değiştirilebilmelidir

4) Structure



5) Implementation

- ResultHandler sınıfı bir Strategy arayüzü üzerinden işlem yapar. Model çıktısına göre sistemin farklı tepkiler üretmesi beklenmektedir. Örneğin bir nesne LCD'ye yazdırılırken, bir diğeri UART'tan gönderilmektedir. Hangi stratejinin kullanılacağı model çıktısına göre belirlenir. setStrategy() fonksiyonu ile dinamik olarak değiştirilir. Kodda doğrudan IStrategy arayüzü tanımlanmasa da, fonksiyonel seviyede switch veya if-else yapılarıyla farklı stratejiler uygulanmaktadır. Stratejilerin çalışma zamanında değiştirilebilmesi ve ortak bir işlem çerçevesinde sunulması, bu desenin uygulandığını göstermektedir.

4. Singleton Pattern

1) Intent

Bir sınıfın sistemde yalnızca tek bir örneğinin olmasını garanti eder ve bu örneğe global erişim sağlar.

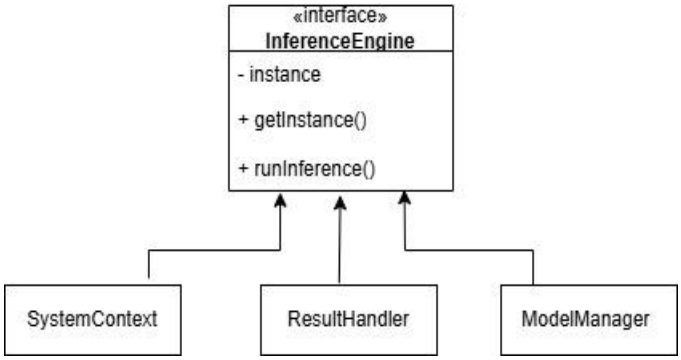
2) Motivation

ESP32 gibi kaynakları sınırlı bir sistemde, UART bağlantısı, I2C arayüzü ya da LCD nesnesi gibi bazı kaynaklar yalnızca bir kez oluşturulmalıdır**.**

3) Applicability

- Tek bir örnek yeterli olduğunda
- Her yerden erişilmesi gerektiğinde
- Kaynakların tekrar tekrar oluşturulması maliyetli olduğunda

4) Structure



5) Implementation

- UART veya LCD nesnesi, ilk çağrıldığında oluşturulur. getInstance() fonksiyonu ile diğer sınıflar hep aynı örneği kullanır. Böylece kaynak israfı ve çakışmalar önlenir.

- ESP32-CAM gibi donanım kısıtlı sistemlerde, özellikle LCD nesnesi, UART bağlantısı veya model çalıştırma motoru gibi bileşenlerin sistemde yalnızca bir örneğinin bulunması gereklidir. Kodda getInstance() gibi klasik Singleton metodu yer almasa da, bu nesneler global olarak yalnızca bir kez tanımlanmakta ve tüm bileşenler bu nesne üzerinden işlem yapmaktadır. Nesnelerin çoğaltılmaması ve her yerden erişilebilir olması, Singleton deseninin uygulanmış olduğunu göstermektedir.