Assignment 1: Design Pattern Explanation-Prepare one page summary explain the MVC(Model-View-Controller) design patterns and its two variants. Use diagram to illustrate their structures and briefly discuss when each variant might be more appropriate to use than the others.

Solution:

MVC(Model-View-Controller) is a software design pattern commonly used for developing user interface that divides the related program logic into three interconnected elements. These elements are:

- the **Model**, the internal representations of information
- the **view**, the interface that presents information to and accepts it from the user
- the **controller**, the software linking the two.

Model:

This component represents the data and the business logic of the application. It encapsulates the data and provides methods to manipulate that data. The model responds to requests for information, performs the necessary actions, and updates itself and, optionally, notifies views of any changes.

View:

The view is responsible for displaying data to the user and capturing user input. It presents the model's data to the user in a meaningful way. Views can be anything from simple user interface elements like buttons and text fields to complex graphical interfaces.

Controller:

The controller acts as an intermediary between the model and the view. It receives user input and decides how to handle it. It updates the model as necessary and may also update the view in response to changes in the model. Essentially, the controller translates user actions into operations on the model.

The Variants of MVC:

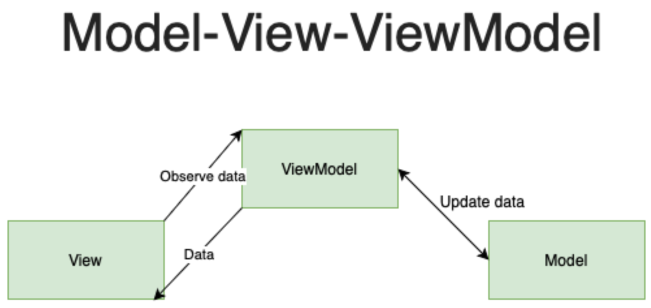
MVC is highly versatile and can be adapted to different scenarios and requirements. MVP (Model-View-Presenter) is a popular variation that sees the presenter as a mediator between the model and the view. The presenter updates the view with data from the model and handles user events and action.

1. **MVVM (Model-View-View Model),**
2. **HMVC (Hierarchical-View-View-Model).**

MVVM (Model-View-View Model):

Model-View-View Model (MVVM) is a design pattern that helps you separate your application logic from the user interface. It is an architectural software design pattern in which the model represents the data, the view is what the user sees and interacts with, and the view model acts as a mediator between the model and view.

MVVM is a great pattern to use when building applications. It helps to separate the business logic from the UI, which makes it much easier to test your application**.**



Components of MVVM:

The Model-View-View Model (MVVM) pattern is a separation of concerns that helps you cleanly separate your application into three main components.

1. **Model**
2. **View**
3. **View Model**

Model:

The model represents the data in your application. The model is generally a domain model, which can be an object-oriented representation of your business domain. This would normally be a collection of objects, properties and methods.

View:

The view represents what the user sees on the screen. View controls and implements visual behaviour of application. It takes keyboard input from the user and provides feedback to them.

View Model:

The view model is responsible for providing data from the model to the view (the user interface). It may also be responsible for handling any interactions between controls on different pages of your application.

The view model is the brain of your application. It’s responsible for providing data from the model (a collection of objects, properties and methods) to the view (the user interface)

Advantage of MVVM:

It allows you to focus on one aspect of a problem at a time, which makes it easier to reason about complex systems. MVVM helps you separate concerns by keeping your UI code separate from your business logic code.

You can replace one piece of code without affecting another. For example, if you want to update

MVVM helps you write reusable code because you can break down your app into smaller chunks that are easier to maintain.

Applications of MVVP

Angular: Angular is a popular front-end framework that implements MVVM architecture.

React: React is a popular JavaScript library that can also implement MVVM architecture using libraries such as MobX or Redux

HMVC (Hierarchical Model-View-Controller):

This is the MVC model repeated over and over again for the parts of your application. And this allows is for parts of the application to be used again elsewhere as each controller can communicate with another controller.

This style of architecture is to modulate the application into smaller more manageable parts that can work independently.

It will also be easier to expand the work to be added to the application due to the capability of controllers calling to each other when a new part is created.

HMVC adds another layer of abstraction by allowing each controller to have its own sub-controllers and sub-views, forming a hierarchy of modules

Advantage of HMVC:

Modularity: HMVC enables programmers to write testable, maintainable code that is modular. Each module may be created individually, and modifications to one module do not affect the other modules.

Code reuse: HMVC makes it simple to reuse code across many modules. Model, View, and Controller components that developers may reuse in other modules can be created.

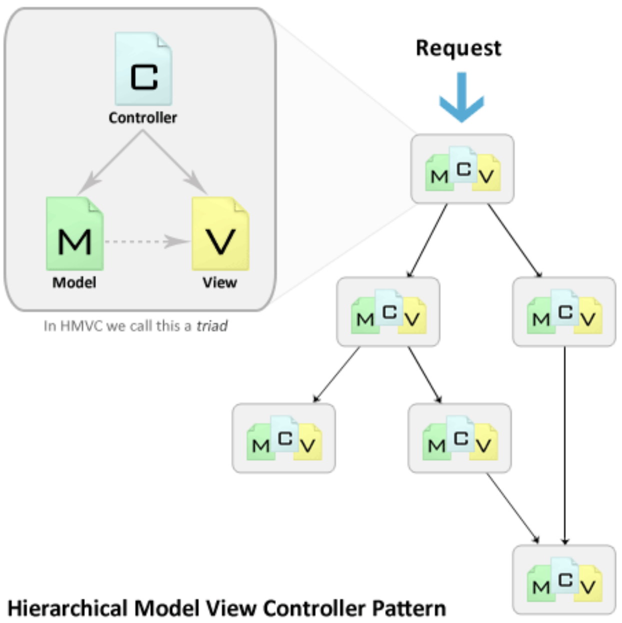
Easy maintenance: By segmenting the code into modules, HMVC makes application maintenance simple. Each module may be tested individually, and modifications to one module do not affect the other modules.

Applications of HMVC:

Dashboard page with various widgets such as charts, tables, and notifications can be a module with its own controller, model, and view.

Blog site with sections like posts, comments, categories, and tags can be a module with its own controller, model, and view that can be reused and integrated into different pages or contexts by using sub-requests and partial views.

Shopping cart system with features like products, orders, payments, or shipping can be a module with its own controller, model, and view that can communicate and coordinate with other modules or the main controller by using sub-requests and partial views.


Controller
Model
View
In HMVC we call this a *triad*
Request
M C V
Hierarchical Model View Controller Pattern

Assignment 2:

Draft a one-page scenario where you apply Micro services Architecture and Event-Driven Architecture to a hypothetical e-commerce platform. Outline how SOLID principles could enhance the design. Use bullet points to indicate how DRY and KISS principles can be observed in this context.

Solution:

A microservices architecture is a type of application architecture where the application is developed as a collection of services. It provides the framework to develop, deploy, and maintain microservices architecture diagrams and services independently.

Service Decomposition: Break down the monolithic e-commerce application into smaller, loosely coupled microservices, each responsible for specific business functionalities such as product catalog, order management, payment processing, and user authentication.

Independent Deployment: Deploy each microservice independently, allowing for faster release cycles and reducing the risk of system-wide failures during updates.

Technology Diversity: Select technology stacks tailored to the specific requirements of each microservice, optimizing performance and flexibility.

API Gateway: Implement an API gateway to manage external client requests and route them to the appropriate microservices, providing a unified interface for external communication.

Benefits of Micro services Architecture:

Microservices architecture offers several benefits compared to traditional monolithic architectures:

Scalability: Microservices allow each component of an application to be scaled independently. This means you can allocate resources specifically to the parts of your application that need them, rather than scaling the entire application at once.

Flexibility and Agility: Microservices enable teams to work on different services simultaneously without interfering with each other. This leads to faster development cycles, as updates and changes can be made to individual services without impacting the entire application.

Resilience: Since microservices are decoupled from each other, failures in one service do not necessarily affect the entire system. This isolation ensures that if one service goes down, the rest of the application can continue to function.

Technology Diversity: Microservices allow for flexibility in technology choices. Each service can be built using the best-suited technology stack for its specific requirements, which can lead to better performance and scalability.

Ease of Maintenance: With microservices, it's easier to maintain and update individual components of an application without affecting the entire system. This modular approach simplifies debugging, testing, and deployment processes.

Improved Fault Isolation: Since each microservice is independent, faults are isolated to specific services, making it easier to identify and address issues.

Enhanced Continuous Deployment: Microservices facilitate continuous deployment and integration practices. Teams can deploy updates to individual services independently, allowing for faster release cycles and reducing the risk associated with large, monolithic deployments.

Scalability: Microservices allow each component of an application to be scaled independently. This means you can allocate resources specifically to the parts of your application that need them, rather than scaling the entire application at once.

Improved Team Productivity: Microservices enable teams to work more autonomously on smaller, focused components of an application. This can lead to increased productivity as teams have clearer ownership and responsibility for their services.

Challenges of Micro services Architecture:

Increased Complexity: Managing a distributed system composed of multiple independent services introduces complexity in terms of deployment, monitoring, and debugging. The intricacies of coordinating interactions between services can be challenging to manage.

Data Management: Maintaining data consistency and ensuring data integrity across multiple services can be challenging. Microservices often lead to data fragmentation, where related data is distributed across different services, making it difficult to manage transactions and ensure data consistency.

Dependency Management: Microservices introduce dependencies between services, and changes in one service can potentially impact other dependent services. Managing dependencies and versioning across multiple services requires careful coordination and communication between development teams.

Testing Complexity: Testing microservices presents unique challenges due to their distributed nature. Comprehensive testing strategies, including integration testing, contract testing, and end-to-end testing, are essential to ensure the reliability and stability of a microservices-based system.

Security: Securing a microservices architecture requires addressing security concerns at multiple layers, including network security, authentication, authorization, and data protection. Ensuring consistent security measures across all services and effectively managing access control can be challenging.

Observing DRY and KISS Principles:

Don't Repeat Yourself (DRY):

- Utilize shared libraries or modules across microservices to avoid duplicating code for common functionalities such as authentication, logging, and error handling.
- Centralize business logic and data validation to ensure consistency and reduce redundancy.

Keep It Simple, Stupid (KISS):

- Favor simplicity and clarity in microservice design, avoiding unnecessary complexity or over-engineering.

- Keep communication between microservices straightforward and well-defined, using standardized protocols and formats to minimize overhead

Event-Driven Architecture to a hypothetical e-commerce platform:

Event driven architecture (EDA) builds on composable commerce’s ability to add features or swap out applications seamlessly, and further accelerates and automates commerce processes by supporting truly real-time, event-driven actions.

With EDA, a time-sensitive ‘event’ (such as a customer placing an order, stock-level notifications, delivery updates, or payments) is used as the trigger for applications and services in your commerce architecture to talk to each other.

EDA enables business flows to be dynamically orchestrated in real time, based on each specific event and its requirements.

With an EDA, services such as the E-Commerce site, order validation, stock management, payment processing, and fulfillment can be from multiple different vendors to suit the organization’s needs. Each service produces and consumes data on new events (such as Jon placing his order), and the event data is filtered and pushed to the relevant services automatically by an event router.

1.Single Responsibility Principle (SRP):

Each microservice within the e-commerce platform should have a single responsibility, focusing on a specific business domain such as product catalog, order management, or user authentication.

By adhering to SRP, microservices become more modular and easier to maintain, as changes to one aspect of the system are less likely to impact other unrelated parts.

2.Open/Closed Principle (OCP):

Microservices should be open for extension but closed for modification. This means that new functionality should be added through extension rather than by changing existing code.

By designing microservices to be open for extension, the e-commerce platform can accommodate future enhancements and changes without disrupting existing functionality, promoting scalability and adaptability.

3.Liskov Substitution Principle (LSP):

Microservices within the same business domain should be interchangeable, meaning that any service can be replaced with another service without altering the correctness of the system.

Adhering to LSP ensures that microservices are designed with consistent interfaces and behaviors, allowing for seamless substitution and integration within the e-commerce platform.

4.Interface Segregation Principle (ISP):

Define clear and minimalistic interfaces between microservices to avoid unnecessary dependencies and promote loose coupling.

By adhering to ISP, microservices can interact with each other through well-defined interfaces, reducing the likelihood of unintended side effects and simplifying system maintenance and evolution.

6.Dependency Inversion Principle (DIP):

Microservices should depend on abstractions rather than concrete implementations, allowing for flexibility and ease of substitution.

By applying DIP, the e-commerce platform can be designed with decoupled components that are easier to test, maintain, and evolve, as dependencies can be swapped out without affecting the overall system behavior.

The benefits of implementing event driven architecture:

- Decoupling allows for faster development cycles, as each service in the architecture can be developed independently from the other.
- Each service can be scaled independently to meet the organization’s specific needs.
- Response times for each service are faster, as they operate independently from each other
- The DRY principle, an acronym for "Don't Repeat Yourself", emphasizes the importance of avoiding code duplication in our applications. The fundamental idea behind DRY is to have a single, authoritative representation of knowledge within our system. By eliminating duplicated code, we can enhance code maintainability, reduce complexity, and minimize the risk of introducing bugs.
- DRY extends beyond literal code duplication. It also encompasses the duplication of knowledge and intent. In other words, it's about expressing the same concept in multiple places, potentially in different ways.
- One way to implement the DRY principle is by utilizing methods. By extracting commonly repeated code into methods, we can centralize its logic and avoid duplicating it across our application.
- The KISS principle, which stands for "Keep It Simple, Stupid", emphasizes the importance of simplicity in software design and development. The goal is to prioritize straightforward solutions over complex ones. By keeping our code simple, we can enhance comprehensibility, usability, and maintainability.
- Adhering to the KISS principle offers several advantages in the context of digital product development.
- Simpler software structures facilitate testing, including automated testing. With less complexity, testing becomes easier and more effective, leading to higher code quality.

Assignment 3: Trends and Cloud Services Overview - Write a three-paragraph report covering: 1) the benefits of serverless architecture, 2) the concept of Progressive Web Apps (PWAs), and 3) the role of AI and Machine Learning in software architecture. Then, in one paragraph, describe the cloud computing service models (SaaS, PaaS, IaaS) and their use cases?

Solution:

Serverless architecture:

A serverless architecture is a way to build and run applications and services without having to manage infrastructure. Your application still runs on servers, but all the server management is done by AWS.

Benefits of Serverless Architecture:

Scalability: Serverless platforms automatically scale resources up or down based on demand. Functions are executed in response to incoming events, allowing for seamless scaling without the need for manual intervention. This enables applications to handle spikes in traffic or workload without provisioning or managing additional infrastructure.

Cost-Efficiency: With serverless computing, you only pay for the resources consumed by your functions or microservices while they are running. Since serverless platforms automatically manage resource allocation and scaling, you can avoid over-provisioning and idle capacity, resulting in cost savings, especially for workloads with unpredictable or intermittent usage patterns.

Reduced Operational Overhead: Serverless architectures eliminate the need for provisioning, configuring, and managing servers or containers. The cloud provider handles infrastructure management, including server maintenance, security patching, and resource allocation. This allows developers to focus more on writing code and less on managing infrastructure, reducing operational complexity and overhead.

Increased Developer Productivity: Serverless platforms abstract away infrastructure concerns, allowing developers to focus on writing and deploying code. Developers can quickly build and deploy functions or microservices without worrying about managing servers, containers, or scaling infrastructure. This accelerates development cycles and enables rapid iteration and experimentation.

Improved Time-to-Market: Serverless architecture enables faster time-to-market for applications by reducing development and deployment cycles. Developers can quickly build and deploy individual functions or microservices, allowing for incremental updates and continuous delivery. This agility enables organizations to respond quickly to changing business requirements and market dynamics.

Limitations of Serverless Computing:

Serverless architecture can be difficult for organizations to secure because of its distributed nature. This makes the architecture more flexible and scalable, which renders traditional security solutions ineffective

Serverless computing also presents visibility issues as developers take the lead, which can result in applications being pushed to production before being addressed by security teams

Over-privileged functions: This is when a function has more permissions than it needs. In this way, it can be abused to attack key systems or exfiltrate or corrupt data.

Poisoning the well**:** This is when attackers try to incorporate malicious code into widely used projects.

Progressive Web App:

A Progressive Web App (PWA) is a type of web application that utilizes modern web technologies to provide a user experience similar to that of native mobile apps. PWAs are designed to work on any device or platform that uses a standards-compliant web browser, including desktops, smartphones, and tablets. They leverage features such as service workers, web app manifests, and responsive design to deliver fast, reliable, and engaging experiences to users.

Key characteristics of Progressive Web Apps include:

Responsiveness: PWAs are built with responsive design principles, ensuring that they look and work well across a variety of devices and screen sizes, from smartphones to desktop computers.

App-Like Experience: PWAs provide an app-like experience to users, with features such as smooth animations, Fullscreen mode, and immersive interactions. They can be launched from the device's home screen and run in a standalone window, giving users the feeling of using a native mobile app.

Reliability: PWAs are designed to be reliable, even in offline or low-connectivity environments. They use service workers to cache resources and enable offline access to content, allowing users to continue using the app even when they're offline or on a slow network.

Performance: PWAs are optimized for performance, with fast loading times and smooth animations. They use techniques such as lazy loading, code splitting, and image optimization to minimize page load times and provide a snappy user experience.

Engagement: PWAs support features such as push notifications, background sync, and add-to-home-screen prompts, which help to increase user engagement and retention. These features allow PWAs to re-engage users and deliver timely updates and notifications.

Security: PWAs are served over HTTPS to ensure the security and integrity of data transmitted between the app and the server. They adhere to web security best practices, such as content security policies and secure cookie settings, to protect against common security threats.

Discoverability: PWAs are discoverable by search engines and can be indexed like any other website. They can also be shared via URLs, making it easy for users to access and share them with others.

The role of AI and Machine Learning in software architecture:

- AI is a branch of computer science which is concerned with the study and creation of computer systems that exhibit some form of intelligence
- AI is a broad area consisting of different fields, from machine vision, expert systems to the creation of machines that can "think".

Characteristics of AI systems:

- Understand a natural language or perceive and comprehend a visual scene.
- Plan sequences of actions to complete a goal.
- May not necessarily imitate human senses and thought processes.
- Capable of performing intelligent tasks effectively and efficiently.

Ways AI & Ml is bound to change how architecture:

AI can reduce the time needed to arrive at a design to mere hours or even minutes. Architects will soon be able to plug into a huge, constantly-updated database of designs and codes, letting technology generate design variations for the client in real-time.

Planning and managing cities, towns, and urban regions is a complex undertaking, and it’s one that’s greatly benefiting from advancements in AI technology. Machine learning can automatically identify the most optimal routes for utilities, which then get updated in real-time as the urban design goes through revisions.

The cloud computing service models (SaaS, PaaS, IaaS) and their use cases:

Cloud computing is the delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the Internet (“the cloud”) to offer faster innovation, flexible resources, and economies of scale.

Types of cloud services: IaaS, PaaS, serverless, and SaaS:

Infrastructure as a Service (IaaS):

Infrastructure as a service is a cloud computing service model by means of which computing resources are supplied by a cloud services provider. The IaaS vendor provides the storage, network, servers, and virtualization. This service enables users to free themselves from maintaining an on-premises data center.

Platform as a Service (PaaS):

Platform as a service (PaaS) is a cloud computing model that provides customers a complete cloud platform—hardware, software and infrastructure—for developing, running and managing applications without the cost, complexity and inflexibility that often comes with building and maintaining that platform on premises.

Software as a Service (SaaS):

Software as a service (SaaS) allows users to connect to and use cloud-based apps over the Internet. Common examples are email, calendaring, and office tools (such as Microsoft Office 365). SaaS provides a complete software solution that you purchase on a pay-as-you-go basis from a cloud service provider.