Class #2

02. Software development principles

Software Architectures

Master in Informatics Engineering

Cláudio Teixeira (claudio@ua.pt)

Agenda

- Cross-cutting concerns
  - Dependency injection and Decorators
- Loose coupling, High cohesion, SOC and SOLID
- Distributed Systems
  - Choreography
  - Security
  - Observability

My O'Reilly Playlist for Software Architecture Course
https://learning.oreilly.com/playlists/e4bbd2ea-5604-419c-8f40-2b0a06e7f8aa

Cross cutting concerns

What are cross-cutting concerns?

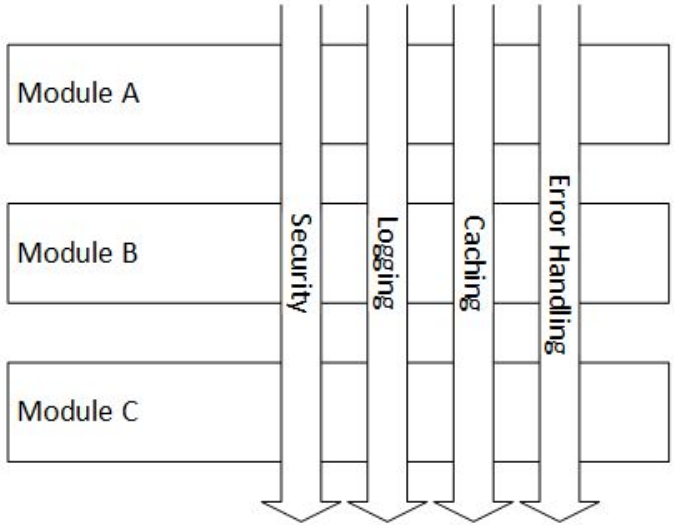
Cross-cutting concerns are aspects of a program that affect multiple components of the system but are not related to the business logic directly.

They represent functionality that is difficult to decompose from the rest of the system using traditional methods of modularization because they span across various points of an application.

Common examples include logging, error handling, data validation, security, transaction management, and performance monitoring.

Implementing these concerns in a modular and maintainable way is crucial for the development of a clean, scalable, and efficient application.

Addressing cross-cutting concerns properly ensures that the core functionality of the application is not cluttered with repetitive and unrelated code, which can lead to issues with maintainability and readability.



Best Practices for Managing Cross-Cutting Concerns

1. Identify Common Concerns: Recognize the functionalities that span multiple modules of your application.
2. Use Aspect-Oriented Programming: Leverage AOP to separate concerns such as logging, security, and transactions from business logic.
3. Employ Dependency Injection: Use DI to inject common functionality, such as logging interfaces, into classes.
4. Implement Middleware: In web applications, use middleware to handle requests and responses for concerns like authentication and error handling.
5. Utilize Attributes: For concerns applicable at the method level, custom attributes can define specific behaviors like caching or validation.

Dependency Injection

Dependency Injection is a design pattern used to implement IoC (Inversion of Control), where the creation and binding of dependent objects are not handled by the objects themselves but by an external entity.

DI allows classes to receive their dependencies from an external source rather than creating them internally.

This promotes loose coupling, enhances testability, and allows for more flexible code management.

Dependency Injection Benefits

Loose Coupling: Dependency Injection significantly reduces the tightness of coupling between software components. By relying on abstractions (interfaces or abstract classes) rather than concrete implementations, DI allows components to communicate without a direct knowledge of each other's construction or lifecycle management. This decoupling not only simplifies the process of swapping out dependencies for alternate implementations but also enhances the modularity of the application, making each component easier to understand, test, and maintain.

Enhanced Testability: DI greatly simplifies the process of unit testing by allowing developers to inject mock objects or stubs in place of real dependencies. This means that classes can be tested in isolation, without relying on their external dependencies, leading to faster, more focused, and reliable tests. By facilitating the creation of test environments that closely mimic controlled conditions, DI ensures that tests can precisely target and validate specific behaviors or logic in the application, thereby improving overall code quality and reliability.

Improved Code Maintenance: With dependencies being injected rather than created within the components themselves, updates or changes to the system's configuration can be made in a centralized location without needing to modify individual components. This abstraction layer provided by DI means that changes to the implementation details of dependencies have minimal impact on the classes that use them. Consequently, the application becomes easier to maintain and evolve over time, as updates, bug fixes, or enhancements require fewer code changes and have reduced risk of unintended side effects.

Dependency Injection Benefits

Increased Flexibility: DI's separation of concerns allows for an unprecedented level of flexibility in application development. Developers can easily switch between different implementations of a given dependency without altering the consuming class's code. This is particularly useful for adapting the application to different environments (e.g., development, staging, production) or integrating third-party services and components. The ability to modify the application's behavior at runtime through configuration changes or dynamic dependency resolution further underscores the agility DI introduces into the software development lifecycle.

Simplified Configuration and Integration: Managing an application's dependencies centrally through a DI container or framework simplifies the configuration and integration of complex systems. DI frameworks typically offer advanced features like automatic dependency resolution, lifecycle management, and configuration-based dependency mapping, which streamline the process of wiring together application components. This centralized approach not only reduces boilerplate code but also ensures that dependency management is consistent and transparent across the application, facilitating easier integration of new components and services.

Removes Hardcoded Dependencies: Dependency Injection eliminates the need for hardcoded dependencies within classes, making the system more modular and flexible. By injecting dependencies, classes do not need to know the concrete implementation of their dependencies. This separation allows dependencies to be changed or updated without altering the dependent class.

Dependency Injection Benefits

Allows Dependencies to be Changed at Runtime or Compile Time: One of the powerful features of DI is the ability to switch dependencies not just at compile time but also at runtime. This flexibility is particularly useful for configurations that depend on different environments (development, testing, production) or for applying changes without needing to recompile the application.

Facilitates Parallel Development: By decoupling classes from their dependencies, DI enables parallel development. Teams can work on different parts of the application simultaneously without waiting for others to complete their components. As long as the interfaces are defined, developers can use mock implementations or stubs to proceed with their work, significantly speeding up the development process.

Programming Against an Interface Rather Than a Class: DI encourages programming against interfaces rather than concrete classes. This approach not only makes the code more resilient to changes but also ensures that any class that implements the interface can be injected without affecting the class's functionality. It underscores the principle of "coding to an interface," making the system more adaptable and extensible.

DI Example

Consider a scenario in a web application where you might want to switch between a database logger and a file logger based on the environment or configuration at runtime

```
// Logger interface
public interface ILogger {
   void Log(string message);}

// Database logger
public class DatabaseLogger : ILogger {
   public void Log(string message) {
       // Logic to log message to database   }}

// File logger
public class FileLogger : ILogger {
   public void Log(string message) {
       // Logic to log message to a file }}

// Client class that uses DI
public class NotificationService {
   private readonly ILogger _logger;

   public NotificationService(ILogger logger) {
       _logger = logger;}

   public void NotifyUser(string message) {
       _logger.Log(message);   }}

// Configuration class to switch dependencies based on the environment
public static class LoggerConfigurator {
   public static ILogger GetConfiguredLogger() {
       Environment.GetEnvironmentVariable("APP_ENVIRONMENT") == "PRODUCTION")?
           return new DatabaseLogger(): return new FileLogger(); }}

// Usage
var logger = LoggerConfigurator.GetConfiguredLogger();
var notificationService = new NotificationService(logger);
```

Patterns of Dependency Injection

Constructor Injection: Dependencies are provided through the class constructor. Ensures that the object is always created with its dependencies.

Property Injection: Dependencies are set through properties or setter methods. Useful when the dependency is optional or when circular dependencies must be resolved.

Method Injection: Dependencies are provided through methods. Allows dependencies to be supplied only when the method is called, providing flexibility.

Constructor Injection

```
public interface ILogger {
   void Log(string message);
}

public class ConsoleLogger : ILogger {
   public void Log(string message) {
       Console.WriteLine(message);
   }
}

public class ProductService {
   private readonly ILogger _logger;

   public ProductService(ILogger logger) {
       _logger = logger;
   }

   public void AddProduct(string name) {
       // Business logic here
       _logger.Log($"Product {name} added.");
   }
}
// Usage
var logger = new ConsoleLogger();
var productService = new ProductService(logger);
```

Property Injection

```
public class CustomerService {
    public ILogger Logger { get; set; } // Property injection

    public void UpdateCustomer(int id) {
        // Business logic here
        Logger?.Log($"Customer {id} updated.");
    }
}

// Usage
var customerService = new CustomerService {
    Logger = new ConsoleLogger()
};
```

Method Injection

```
public class OrderService {
   public void ProcessOrder(int orderId, ILogger logger) {
       // Business logic here
       logger.Log($"Order {orderId} processed.");
   }
}

// Usage
var orderService = new OrderService();
orderService.ProcessOrder(123, new ConsoleLogger());
```

Decorator Pattern

The Decorator pattern is a structural design pattern that allows for the dynamic addition of behaviors to objects without modifying their existing classes.

It provides an alternative to subclassing for extending functionality, using composition over inheritance.

By wrapping an object within an object of the same interface, decorators can add new responsibilities at runtime, making this pattern highly flexible and versatile.

Decorator Pattern

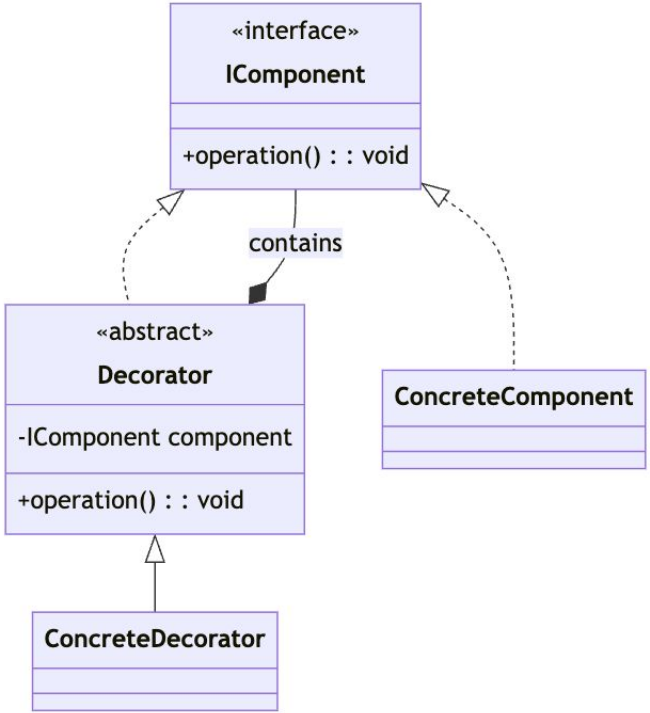
Key Concepts

Component: An interface for objects that can have responsibilities added to them dynamically.

ConcreteComponent: A class implementing the Component interface, representing objects to which additional responsibilities can be attached.

Decorator: An abstract class implementing the Component interface and holding a reference to a Component object. It delegates calls to the wrapped object and may add additional behavior before or after forwarding the call.

ConcreteDecorator: Classes that extend the Decorator class, implementing additional behaviors that can be added to ConcreteComponents.



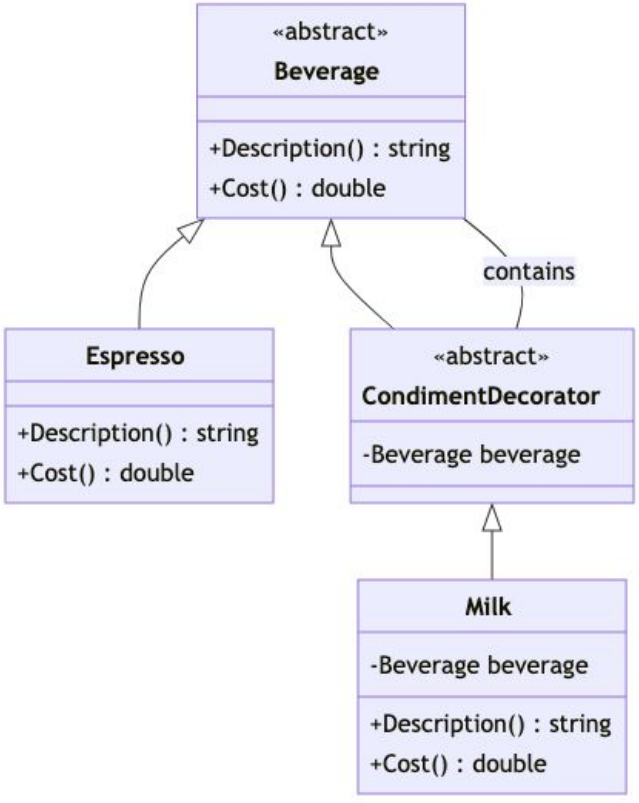
Decorator Pattern
Benefits

Enhanced Flexibility: Unlike inheritance, the Decorator pattern allows for the dynamic addition or removal of behaviors to or from objects without affecting other instances of the same class.

Avoids Class Proliferation: It helps manage class explosion by allowing for functionality extension without creating numerous subclasses.

Single Responsibility Principle: Decorators focus on adding a single responsibility to objects, keeping the system design clean and straightforward.

Imagine a simple coffee ordering system where you can dynamically add extras (like milk, sugar, etc.) to your coffee order without creating a subclass for every combination.



Example

```
//Component (Beverage)
public abstract class Beverage {
   public abstract string Description { get; }
   public abstract double Cost();}

//ConcreteComponent (Espresso)
public class Espresso : Beverage {
   public override string Description => "Espresso";

   public override double Cost() {
       return 1.99;}
}

//Decorator (CondimentDecorator)
public abstract class CondimentDecorator : Beverage {
   public abstract override string Description { get; }
}

//ConcreteDecorator (Milk)
public class Milk : CondimentDecorator {
   private readonly Beverage _beverage;
   public Milk(Beverage beverage) {
       _beverage = beverage;}
   public override string Description => _beverage.Description + ", Milk";
   public override double Cost() {
       return _beverage.Cost() + 0.50;}
}

//Using the decorator
Beverage beverage = new Espresso();
Console.WriteLine($"{beverage.Description} ${beverage.Cost()}");

beverage = new Milk(beverage); // Adding milk
Console.WriteLine($"{beverage.Description} ${beverage.Cost()}");
```

Coupling

Coupling

Coupling in software architecture refers to the degree of interdependence between software modules or components. It indicates how closely connected or reliant one module is on another. In general, lower coupling leads to more modular, maintainable, and scalable systems, while higher coupling can result in code that is harder to understand, modify, and extend. Coupling types (from higher coupling to lower, looser, coupling):

1. Content Coupling
2. Common Coupling
3. External Coupling
4. Control Coupling
5. Stamp Coupling
6. Data Coupling
7. External Coupling
8. Message Coupling
9. No Coupling

Content coupling

Content coupling, also known as control coupling, occurs when one module modifies or relies on the internal data of another module.

This type of coupling is undesirable as it exposes the internal details of one module to another, leading to dependencies on implementation details.

Example: A function modifying the internal state of an object passed as a parameter.

```
// Content Coupling Example
public class CustomerService {
 private Customer customer = new Customer(); // Content
coupling with Customer class
  public void UpdateCustomerAddress(string address) {
     customer.Address = address; // Modifying internal state
of Customer object
     // Address update logic...
 }
}

public class Customer {
 public string Address { get; set; }
}
```

Common Coupling

Common coupling occurs when multiple modules share a global data structure, such as a global variable or a shared file.

Changes to the shared data structure can impact multiple modules, making it challenging to predict the effects of modifications.

Example: Several modules accessing and modifying a global configuration file.

```
// Common Coupling Example
public static class AppConfig {
   public static string ConnectionString = "Data Source=database.db"; //
Common coupling with global variable
}

public class DatabaseService {
   public void Connect() {
       string connectionString = AppConfig.ConnectionString; //
Accessing global variable
       // Database connection logic...
   }
}
```

External Coupling

External coupling occurs when two modules depend on the same external entity, such as a database or an external library.

Changes to the external entity can affect both modules, but they are not directly dependent on each other's internal details. The main challenge is on reducing, rather than avoiding its use.

Example: Two modules accessing the same database tables or using the same third-party library.

```
// External Coupling Example
using Newtonsoft.Json; // External dependency

public class JsonParser {
    public void Parse(string json) {
        var parsedObject = JsonConvert.DeserializeObject(json); //
External library usage
        // Parsing logic...
    }
}
```

Control Coupling

Control coupling occurs when one module controls the behavior of another module by passing control information, such as flags or parameters.

This type of coupling can lead to complex control flows and make the code harder to understand.

Example: A function calling another function and passing a control flag to determine its behavior.

```
// Control Coupling Example
public class PaymentService {
    public void ProcessPayment(bool isOnlinePayment) {
        if (isOnlinePayment) {
            // Process online payment...
        } else {
            // Process offline payment...
        }
    }
}
```

Stamp Coupling
(data-structured coupling)

Stamp coupling occurs when modules share a composite data structure and only use a portion of it.

This type of coupling can result in unnecessary dependencies and increase the risk of errors when modifying the shared data structure.

Example: Two modules sharing a large data object but only using specific fields or attributes.

```
// Stamp Coupling Example
public class UserDetails {
    public string Name { get; set; }
    public string Email { get; set; }
    public string Address { get; set; }
    // Other user details...
}
public class UserProcessor {
    public void ProcessUser(UserDetails userDetails) {
        // Uses only the Address from userDetails
        Console.WriteLine($"Processing user at address:
{userDetails.Address}");
        // Process logic...
    }
}

public class NewsletterService {
    public void SendNewsletter(UserDetails userDetails) {
        // Uses only the Email from userDetails
        Console.WriteLine($"Sending newsletter to: {userDetails.Email}");
        // Sending logic...
    }
}
```

Data Coupling

Data coupling occurs when modules interact with each other through data passed in parameters.

In data coupling, modules do not depend on each other's internal workings; instead, they communicate strictly through the data they exchange.

This form of coupling is desirable because it reduces dependencies between modules, making the system easier to understand and modify.

Example: A function calling another function and passing parameters or receiving return values to exchange data.

```
public class OrderProcessor
{
    public void ProcessOrder(int orderId, decimal orderTotal)
    {
        // Processing the order based on orderId and orderTotal
        Console.WriteLine($"Processing order #{orderId} with total
{orderTotal}");
    }
}
public class OrderService
{
    private readonly OrderProcessor _orderProcessor = new
OrderProcessor();

    public void SubmitOrder(int orderId, decimal orderTotal)
    {
        // Submitting the order involves processing it
        _orderProcessor.ProcessOrder(orderId, orderTotal);
    }
}
```

Message Coupling

Message coupling occurs when modules communicate by sending messages to each other, typically through message queues or event-driven mechanisms.

In this type of coupling, modules are decoupled in terms of implementation details, as they interact only through predefined message formats.

Example: Two modules communicating asynchronously by publishing and subscribing to events using a message broker like RabbitMQ or Kafka.

```
// Message Coupling Example
using System;

public class MessagePublisher {
    public event EventHandler<MessageEventArgs> MessagePublished;

    public void PublishMessage(string message) {
        OnMessagePublished(new MessageEventArgs(message));
    }

    protected virtual void OnMessagePublished(MessageEventArgs e) {
        MessagePublished?.Invoke(this, e);
    }
}

public class MessageSubscriber {
    public void Subscribe(MessagePublisher publisher) {
        publisher.MessagePublished += Publisher_MessagePublished;
    }

    private void Publisher_MessagePublished(object sender,
MessageEventArgs e) {
        Console.WriteLine($"Received message: {e.Message}");
    }
}

public class MessageEventArgs : EventArgs {
    public string Message { get; }

    public MessageEventArgs(string message) {
        Message = message;
    }
}
```

No Coupling :)

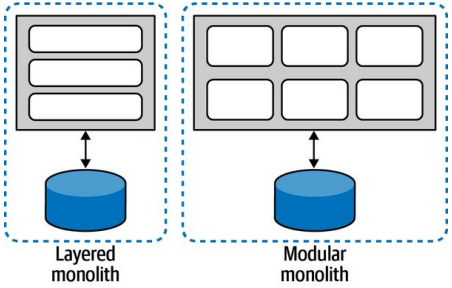
No coupling, also known as zero coupling or independent coupling, represents the ideal scenario where modules have no dependencies on each other.

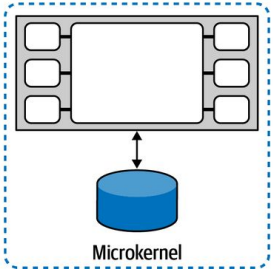
Modules operate entirely independently, with no direct or indirect interactions.

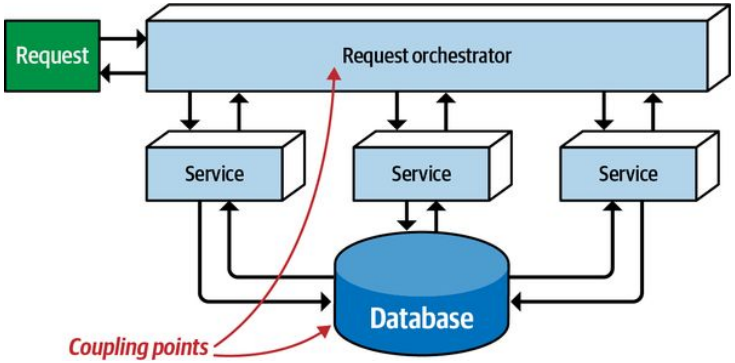
Achieving no coupling is rare in practice but represents the ultimate goal of designing loosely coupled systems.

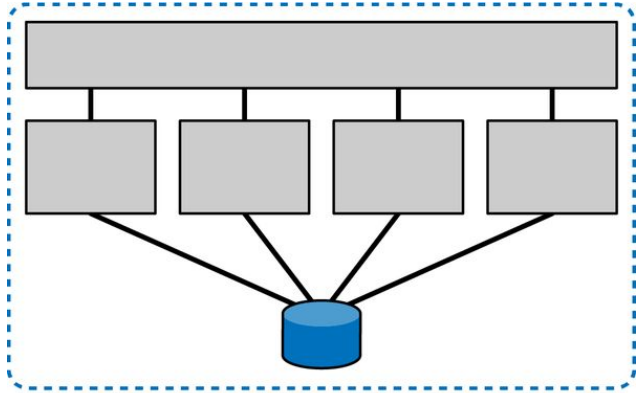
Example: Two standalone modules that perform separate tasks without any shared resources or dependencies.

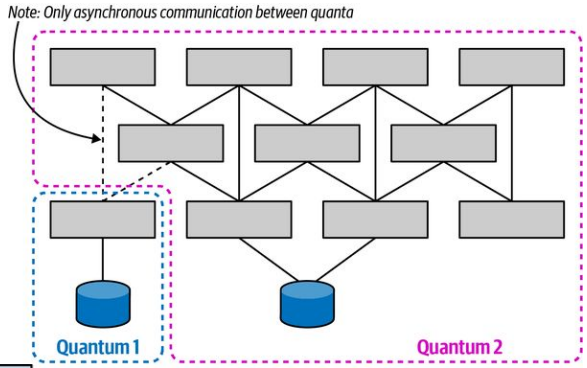
```
// No Coupling Example
// Two independent classes with no dependencies between them
public class ClassA {
    public void MethodA() {
        // Method implementation...
    }
}

public class ClassB {
    public void MethodB() {
        // Method implementation...
    }
}
```

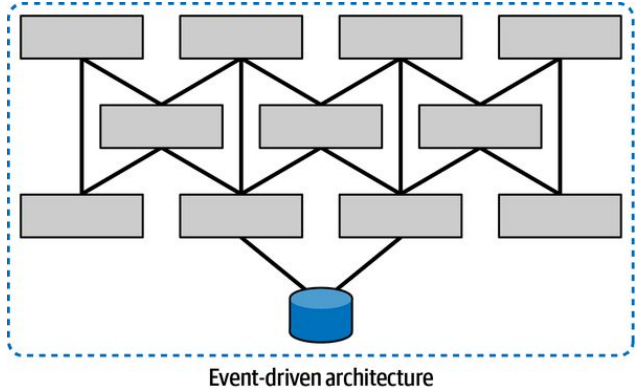


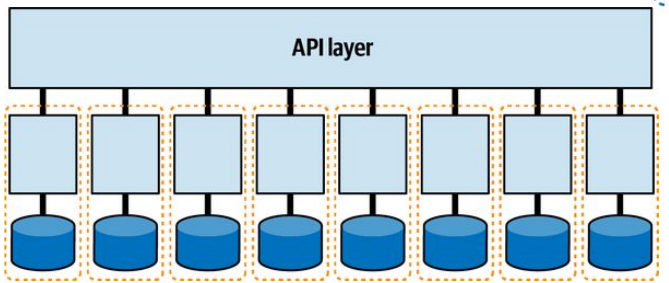












A few schematics on organizing couplings

Cross cutting concerns challenge

40 minutes
20 min class discussion

https://docs.google.com/document/d/1Apq57nWqyyOo2ReJOdHpMMQRVWRrUwljPL17JQ1lY6c/edit?tab=t.0

Exercise Solo:
Deploying Observability

60 minutes
10 min class discussion

https://docs.google.com/document/d/10VDvDJZLJkAXhWVBG2sBTn7_QiOZRDhdikUhY5y0KpI/edit?usp=sharing

Bibliography

- (Garlan, 1992) – David Garlan, Mary Shaw. “An Introduction to Software Architecture”
- (Mens, T. , Demeyer, S. , 2008) – Tom Mens and Serge Demeyer, “Software Evolution”, ISBN: 978-3-540-76440-3, http://www.springerlink.com/content/978-3-540-76439-7/#section=200232&page=1
- (Bass et al, 1998) – L. Bass, P. Clements, R. Kazman (1998). Software Architecture in Practice. Reading, MA: Addison Wesley Longman, Inc.
- (Garlan et al, 1993) – D. Garlan, M. Shaw (1993). "An Introduction to Software Architecture." In V. Ambriola, G. Tortora, eds., Advances in Software Engineering and Knowledge Engineering, Vol. 2, pp. 1—39. New Jersey: World Scientific Publishing Company.
- Solution Architecture Patterns for Enterprise: A Guide to Building Enterprise Software Systems
- https://camunda.com/blog/2023/02/orchestration-vs-choreography/
- https://learn.microsoft.com/en-us/azure/architecture/patterns/choreography