Corso C# – Design Pattern, SOLID, IoC, DI e Microservizi

Design Pattern GoF

Slide 1: Introduzione ai Design Pattern

I **design pattern** (schemi di progettazione) sono soluzioni **riutilizzabili** a problemi progettuali **ricorrenti** nello sviluppo software. Introdotti dalla Gang of Four (GoF), forniscono strutture comprovate per risolvere sfide comuni, migliorando la qualità e la manutenibilità del codice. Ogni pattern descrive un problema specifico e la soluzione progettuale per affrontarlo, in un certo **contesto**.

I pattern si suddividono in tre categorie principali: creazionali, strutturali e comportamentali. Applicare un design pattern appropriato consente di evitare codice duplicato o architetture rigide, favorendo un **accoppiamento debole** e un codice più flessibile. Nei prossimi slide vedremo alcuni pattern GoF importanti (es. *Singleton, Factory, Strategy, Observer, Decorator, Adapter*) attraverso spiegazioni ed esempi di codice.

```
// Esempio (problema ricorrente senza pattern):
if (formato == "PDF")
    GeneraReportPDF(dati);
else if (formato == "HTML")
    GeneraReportHTML(dati);
// Questa logica condizionale può crescere e diventare difficile da gestire.
```

Slide 2: Singleton Pattern

Il **Singleton** è un pattern creazionale che garantisce che una classe abbia **una singola istanza** globale e fornisce un punto di accesso centralizzato a tale istanza. È utile quando occorre un oggetto unico condiviso, ad esempio un logger o una gestione centralizzata di configurazione. Il Singleton previene la creazione arbitraria di oggetti, controllando lui stesso la propria istanza unica.

Gli elementi chiave di questo pattern sono un **costruttore privato** (per impedire istanziazioni esterne) e un membro accessibile pubblicamente (metodo o proprietà **statica**) che restituisce l'istanza unica, creandola alla prima richiesta. Bisogna fare attenzione in ambienti multi-thread per gestire correttamente l'accesso concorrente all'istanza, altrimenti si rischiano condizioni di gara.

```
class ConfigurationManager {
    private static ConfigurationManager _istanza;
    private ConfigurationManager() { /* costruttore privato */ }

    public static ConfigurationManager Istanza {
        get {
            if (_istanza == null)
```

```
                _istanza = new ConfigurationManager();  // creazione lazy
della singola istanza
            return _istanza;
        }
    }

    public void CaricaConfigurazione() {
        Console.WriteLine("Configurazione caricata.");
    }
}
// Utilizzo del Singleton:
ConfigurationManager.Istanza.CaricaConfigurazione();
```

Slide 3: Factory Method Pattern

Il pattern **Factory Method** è un pattern creazionale che centralizza la **creazione di oggetti** attraverso un metodo dedicato (factory). L'idea è di fornire un'interfaccia comune per istanziare classi, lasciando alle sottoclassi o a una factory il compito di decidere quale classe concreta creare. In questo modo si **disaccoppia** il codice client dalla logica di istanziazione concreta, facilitando l'estensione per nuovi tipi senza modificare il codice esistente.

Grazie al Factory Method, per aggiungere un nuovo tipo di prodotto basta estendere la factory o aggiungere una nuova sottoclasse che override il metodo factory, invece di cambiare condizioni nel codice client. Ciò rende il sistema **aperto all'estensione ma chiuso alle modifiche** (principio OCP). Ad esempio, in un generatore di documenti possiamo usare una factory per creare documenti PDF o HTML a seconda del formato richiesto.

```
// Definizione dei prodotti:
abstract class Documento { public abstract void Genera(); }
class PdfDocumento : Documento { public override void Genera() {
Console.WriteLine("Genera PDF"); } }
class HtmlDocumento : Documento { public override void Genera() {
Console.WriteLine("Genera HTML"); } }

// Factory che crea il prodotto appropriato:
static class DocumentFactory {
    public static Documento CreaDocumento(string tipo) {
        if (tipo == "PDF") return new PdfDocumento();
        else if (tipo == "HTML") return new HtmlDocumento();
        throw new ArgumentException("Tipo documento sconosciuto");
    }
}

// Utilizzo della factory:
Documento doc = DocumentFactory.CreaDocumento("PDF");
doc.Genera();  // output: "Genera PDF"
```

Slide 4: Strategy Pattern

Lo **Strategy Pattern** è un pattern comportamentale che permette di definire una famiglia di algoritmi, incapsularli in classi separare e renderli **intercambiabili**. Il client può scegliere a **runtime** quale algoritmo (strategia) utilizzare, senza modificare il proprio codice. Questo elimina lunghe istruzioni condizionali basate sul tipo di algoritmo, delegando la variazione del comportamento alle diverse implementazioni dell'interfaccia strategica.

Il vantaggio principale è la facilità con cui si possono aggiungere nuove strategie: basta creare una nuova classe che implementa l'interfaccia senza toccare il codice esistente, rispettando il principio **open/closed**. Ad esempio, per calcolare un prezzo con sconti diversi, possiamo definire più strategie di calcolo (nessuno sconto, sconto percentuale, sconto fisso) e scegliere quella appropriata al momento del pagamento.

```
// Interfaccia strategica:
interface ISconto { decimal ApplicaSconto(decimal importo); }

// Implementazioni concrete della strategia:
class NessunoSconto : ISconto {
    public decimal ApplicaSconto(decimal importo) => importo;  // Nessun
cambiamento
}
class ScontoPercentuale : ISconto {
    private decimal _percentuale;
    public ScontoPercentuale(decimal percentuale) { _percentuale =
percentuale; }
    public decimal ApplicaSconto(decimal importo) => importo * (1 -
_percentuale);
}

// Contesto che utilizza una strategia:
class Carrello {
    private ISconto _strategiaSconto;
    public Carrello(ISconto strategiaIniziale) { _strategiaSconto =
strategiaIniziale; }

    public void ImpostaStrategia(ISconto nuovaStrategia) {
        _strategiaSconto =
nuovaStrategia;  // permette di cambiare strategia a runtime
    }
    public decimal CalcolaTotale(decimal importoBase) {
        return _strategiaSconto.ApplicaSconto(importoBase);
    }
}

// Esempio d'uso:
Carrello carrello = new Carrello(new NessunoSconto());
Console.WriteLine(carrello.CalcolaTotale(100));  // 100, nessuno sconto
carrello.ImpostaStrategia(new ScontoPercentuale(0.1m));
Console.WriteLine(carrello.CalcolaTotale(100));  // 90, con 10% di sconto
```

Slide 5: Observer Pattern

L'**Observer Pattern** è un pattern comportamentale che definisce una relazione di **dipendenza uno-a-molti** tra oggetti, tale che quando un oggetto (Subject) cambia stato, tutti i suoi **osservatori** vengono notificati e aggiornati automaticamente. Questo consente di mantenere **disaccoppiati** il soggetto e gli osservatori: il Subject non conosce i dettagli degli observer, sa solo che devono essere informati delle variazioni.

Questo pattern è utile per implementare sistemi di **evento** e sottoscrizione (pub-sub). Un esempio classico è un modello di pubblicazione di notizie: quando c'è una nuova notizia (Subject), tutti gli iscritti (Observers) ricevono una notifica. In C#, il pattern Observer può essere implementato manualmente con interfacce oppure sfruttando il meccanismo degli **eventi** del linguaggio, che è un observer built-in.

```
// Implementazione semplice del pattern Observer:
interface IObserver { void Aggiorna(string messaggio); }

class Subject {
    private List<IObserver> _osservatori = new List<IObserver>();

    public void Registra(IObserver obs) => _osservatori.Add(obs);
    public void Rimuovi(IObserver obs) => _osservatori.Remove(obs);

    public void Notifica(string msg) {
        foreach (var obs in _osservatori) {
            obs.Aggiorna(msg);
        }
    }
}

class OsservatoreConcreto : IObserver {
    private string _nome;
    public OsservatoreConcreto(string nome) { _nome = nome; }
    public void Aggiorna(string messaggio) {
        Console.WriteLine($"{_nome} ricevuto aggiornamento: {messaggio}");
    }
}

// Esempio d'uso:
Subject fonte = new Subject();
var obs1 = new OsservatoreConcreto("Observer1");
var obs2 = new OsservatoreConcreto("Observer2");
fonte.Registra(obs1);
fonte.Registra(obs2);
fonte.Notifica("Nuovo evento disponibile!");
// Output:
// Observer1 ricevuto aggiornamento: Nuovo evento disponibile!
// Observer2 ricevuto aggiornamento: Nuovo evento disponibile!
```

Slide 6: Decorator Pattern

Il **Decorator Pattern** è un pattern strutturale che consente di **estendere funzionalità** di un oggetto **dinamicamente**, avvolgendolo con un oggetto decoratore. Il decoratore implementa la stessa interfaccia dell'oggetto originale e **contiene** al suo interno l'oggetto da decorare, delegandogli le operazioni di base ed eventualmente aggiungendo comportamenti extra prima o dopo tali operazioni. In questo modo si possono combinare decoratori in cascata per aggiungere più comportamenti.

Questo approccio tramite **composizione** è più flessibile dell'ereditarietà, perché permette di aggiungere responsabilità a singoli oggetti senza modificare la classe originale o influire sugli altri oggetti. Ad esempio, se abbiamo un'interfaccia `INotifica` con un metodo `Invia()`, possiamo avere implementazioni base (invio email) e decoratori che aggiungono funzionalità (invio SMS, log su file) in modo intercambiabile e configurabile.

```
interface INotifica { void Invia(string msg); }

class NotificaBase : INotifica {
    public void Invia(string msg) {
        Console.WriteLine($"Email inviata: {msg}");
    }
}

// Decoratore astratto che implementa INotifica e contiene una INotifica
abstract class NotificaDecorator : INotifica {
    protected INotifica _wrappee;
    public NotificaDecorator(INotifica wrappee) { _wrappee = wrappee; }
    public virtual void Invia(string msg) {
        _wrappee.Invia(msg);  // comportamento base delegato
    }
}

// Decoratore concreto che aggiunge invio SMS
class NotificaConSms : NotificaDecorator {
    public NotificaConSms(INotifica wrappee) : base(wrappee) { }
    public override void Invia(string msg) {
        base.Invia(msg);
        Console.WriteLine($"SMS inviato: {msg}");  // funzionalità aggiunta
    }
}

// Utilizzo del decoratore:
INotifica notifier = new NotificaConSms(new NotificaBase());
notifier.Invia("Promozione disponibile!");
// Output:
// Email inviata: Promozione disponibile!
// SMS inviato: Promozione disponibile!
```

Slide 7: Adapter Pattern

L'**Adapter Pattern** è un pattern strutturale che permette a classi con **interfacce incompatibili** di collaborare tra loro. L'adapter agisce come un **traduttore**: espone l'interfaccia attesa dal client (Target) e internamente **adatta** le chiamate delegandole a un oggetto esistente (Adaptee) con un'interfaccia diversa. In sostanza, converte l'interfaccia di una classe in un'altra più adatta a ciò di cui ha bisogno il client.

Questo pattern è utile quando si vuole riutilizzare codice esistente che non può essere modificato, integrandolo in un nuovo sistema. Ad esempio, se il nostro codice aspetta un'interfaccia `ITarget` con un metodo `Richiesta()`, ma disponiamo di una classe esistente con un metodo diverso, possiamo creare un Adapter che implementa `ITarget` e chiama internamente il metodo della classe esistente, consentendo il **riutilizzo** senza cambiare l'esistente.

```
// Interfaccia target prevista dal client:
interface ITarget {
    string Richiesta();
}

// Classe esistente con interfaccia incompatibile (Adaptee):
class ServizioEsistente {
    public string SpecificRequest() {
        return "Dati dal servizio esistente";
    }
}

// Adapter che rende compatibile ServizioEsistente con ITarget:
class ServizioAdapter : ITarget {
    private ServizioEsistente _adaptee;
    public ServizioAdapter(ServizioEsistente adaptee) {
        _adaptee = adaptee;
    }
    public string Richiesta() {
        // Adatta la chiamata all'interfaccia attesa dal client
        return _adaptee.SpecificRequest();
    }
}

// Utilizzo dell'Adapter:
ServizioEsistente servizio = new ServizioEsistente();
ITarget target = new ServizioAdapter(servizio);
Console.WriteLine(target.Richiesta());  // Il client usa ITarget, ma
internamente chiama ServizioEsistente
```

Slide 8: Abstract Factory Pattern

L'**Abstract Factory** è un pattern creazionale che fornisce un'interfaccia per creare **famiglie di oggetti correlati** fra loro, senza specificare le classi concrete. A differenza del Factory Method (che crea un singolo prodotto), l'Abstract Factory produce vari oggetti, garantendo che siano compatibili o parte di

una famiglia coerente. Questo pattern è utile quando occorre garantire che vari componenti (es. pulsanti, finestre) abbiano uno stile o implementazione coordinata (es. tema Windows vs tema macOS).

Il beneficio è poter **astrarre la creazione di gruppi** di oggetti tra loro compatibili. Il client codifica contro l'interfaccia della factory astratta e non conosce le classi concrete effettivamente create. L'aggiunta di una nuova famiglia di prodotti (ad es. un nuovo tema) richiede l'implementazione di una nuova factory concreta, senza toccare il codice client.

```
// Prodotti astratti:
abstract class Bottone { public abstract void Disegna(); }
abstract class Checkbox { public abstract void Disegna(); }

// Implementazioni concrete dei prodotti per Windows:
class WinBottone : Bottone { public override void Disegna() {
Console.WriteLine("Disegna bottone stile Windows"); } }
class WinCheckbox : Checkbox { public override void Disegna() {
Console.WriteLine("Disegna checkbox stile Windows"); } }

// Implementazioni concrete dei prodotti per macOS:
class MacBottone : Bottone { public override void Disegna() {
Console.WriteLine("Disegna bottone stile macOS"); } }
class MacCheckbox : Checkbox { public override void Disegna() {
Console.WriteLine("Disegna checkbox stile macOS"); } }

// Factory astratta:
abstract class GUIFactory {
    public abstract Bottone CreaBottone();
    public abstract Checkbox CreaCheckbox();
}

// Factory concrete per ciascuna famiglia:
class WinFactory : GUIFactory {
    public override Bottone CreaBottone() => new WinBottone();
    public override Checkbox CreaCheckbox() => new WinCheckbox();
}
class MacFactory : GUIFactory {
    public override Bottone CreaBottone() => new MacBottone();
    public override Checkbox CreaCheckbox() => new MacCheckbox();
}

// Utilizzo:
GUIFactory factory = new WinFactory();
Bottone btn = factory.CreaBottone();
Checkbox cb = factory.CreaCheckbox();
btn.Disegna();    // Disegna bottone stile Windows
cb.Disegna();     // Disegna checkbox stile Windows
```

Slide 9: Facade Pattern

Il **Facade Pattern** è un pattern strutturale che fornisce un'interfaccia **unificata e semplificata** a un insieme di interfacce di un sottosistema complesso. In pratica, la Facade è una classe che incapsula la complessità di varie operazioni sottostanti, esponendo metodi di alto livello facili da usare per il client. Ciò riduce le dipendenze del codice client dai dettagli del sistema interno e promuove un utilizzo più semplice e chiaro.

Questo pattern è utile quando si ha a che fare con sistemi legacy o librerie con molte classi e API: creare una Facade consente di **semplificare l'uso** comune del sistema, mantenendo comunque la possibilità di accedere alle funzionalità granulari se necessario. Ad esempio, in un sottosistema che gestisce ordini, pagamenti e spedizioni, si potrebbe avere una Facade `OrderProcessor` che internamente chiama i vari servizi nell'ordine corretto, offrendo al chiamante un singolo metodo `ProcessaOrdine()`.

```
// Sottosistema complesso:
class SistemaPagamento { public void ProcessaPagamento(decimal importo) =>
Console.WriteLine("Pagamento elaborato"); }
class SistemaSpedizioni { public void AvviaSpedizione(int ordineId) =>
Console.WriteLine("Spedizione avviata"); }

// Facade che semplifica l'interazione col sottosistema:
class OrderFacade {
    private SistemaPagamento _pag = new SistemaPagamento();
    private SistemaSpedizioni _spd = new SistemaSpedizioni();

    public void CompletaOrdine(int ordineId, decimal totale) {
        _pag.ProcessaPagamento(totale);
        _spd.AvviaSpedizione(ordineId);
        Console.WriteLine("Ordine #"+ordineId+" completato con successo");
    }
}

// Utilizzo della Facade:
OrderFacade ordineFacade = new OrderFacade();
ordineFacade.CompletaOrdine(42, 149.99m);
// Output:
// Pagamento elaborato
// Spedizione avviata
// Ordine #42 completato con successo
```

Slide 10: Command Pattern

Il **Command Pattern** è un pattern comportamentale che **incapsula una richiesta come oggetto**, permettendo di parametrizzare i client con diversi richieste, accodare o registrare le operazioni e supportare operazioni annullabili (undo). In sostanza, si crea un'interfaccia per i comandi con un metodo esegui (`Esegui()`), implementata da varie classi comando, ciascuna contenente la logica per compiere una determinata azione. Un *Invoker* conserva i comandi e li invoca quando necessario.

Questo pattern **disaccoppia** il chiamante dall'esecutore effettivo dell'azione. Ad esempio, in un'applicazione GUI un pulsante "Annulla" può attivare il metodo `Undo()` dell'ultimo comando

eseguito. I comandi possono inoltre essere composti in macro, messi in coda, salvati per il replay ecc. L'uso di oggetti comando rende più facile estendere il sistema con nuove operazioni senza modificare i pulsanti o gli invoker.

```
// Interfaccia Command con operazione da eseguire:
interface ICommand { void Esegui(); }

// Comandi concreti:
class AccendiLuce : ICommand {
    public void Esegui() => Console.WriteLine("Luce accesa");
}
class SpegniLuce : ICommand {
    public void Esegui() => Console.WriteLine("Luce spenta");
}

// Invoker che può memorizzare ed eseguire un comando:
class Telecomando {
    private ICommand _comando;
    public void ImpostaComando(ICommand comando) { _comando = comando; }
    public void PremiPulsante() {
        _comando.Esegui();
    }
}

// Utilizzo:
Telecomando rc = new Telecomando();
rc.ImpostaComando(new AccendiLuce());
rc.PremiPulsante();  // Output: "Luce accesa"
rc.ImpostaComando(new SpegniLuce());
rc.PremiPulsante();  // Output: "Luce spenta"
```

Slide 11: Vantaggi dei Design Pattern

In sintesi, i design pattern forniscono soluzioni **riutilizzabili** e collaudate che migliorano la struttura del codice. Applicando un pattern appropriato, il codice diventa più **manutenibile**, leggibile ed estensibile. I pattern favoriscono un lessico comune tra sviluppatori: ad esempio, dire "uso un Observer qui" comunica immediatamente l'intenzione progettuale, facilitando la **comunicazione** nel team e le revisioni del codice.

Bisogna tuttavia usare i pattern con giudizio: introdurli quando risolvono un problema reale di design e non forzare l'uso di un pattern se non necessario. Ogni pattern ha un contesto di applicabilità; conoscerli aiuta a costruire un'**architettura solida** e ad evitare soluzioni ad-hoc fragili. In pratica quotidiana, i pattern come quelli visti (Singleton, Factory, etc.) aiutano a ridurre duplicazione di codice e ad **organizzare meglio** le responsabilità nel sistema.

```
/* Prima (senza pattern): gestione pagamenti con logica condizionale */
if (metodoPagamento == "Carta") ProcessaCarta(dati);
else if (metodoPagamento == "PayPal") ProcessaPayPal(dati);
// Aggiungere un nuovo metodo di pagamento richiede modifica di questo codice
```

```
esistente.

/* Dopo (con Strategy Pattern): */
IPagamento metodo = new PagamentoCarta();      // scelta della strategia di
pagamento
cassa.ProcediPagamento(metodo);
// Nuovi metodi di pagamento possono essere aggiunti implementando
IPagamento, senza alterare il codice di cassa.
```

Principi SOLID

Slide 12: Principi SOLID – Introduzione

I principi **SOLID** sono cinque linee guida fondamentali per una progettazione orientata agli oggetti **manutenibile** e **flessibile**. Coniati da *Robert C. Martin (Uncle Bob)*, aiutano a evitare i problemi più comuni di design (rigidità, fragilità, accoppiamento eccessivo) e facilitano l'estensione e la comprensione del codice. SOLID è un acronimo che sta per: - **S**: Single Responsibility Principle (*principio della singola responsabilità*), - **O**: Open/Closed Principle (*principio aperto/chiuso*), - **L**: Liskov Substitution Principle (*principio di sostituzione di Liskov*), - **I**: Interface Segregation Principle (*principio di segregazione delle interfacce*), - **D**: Dependency Inversion Principle (*principio di inversione delle dipendenze*).

Seguire i principi SOLID porta a classi con responsabilità ben definite, con **basso accoppiamento** e alta coesione. Questo rende il sistema più adattabile al cambiamento e facilita il testing. Nei prossimi slide vedremo ciascun principio SOLID, con esempi in C# di violazione e poi di applicazione corretta del principio, per evidenziarne l'effetto sul design.

```
// Esempio di classe con problemi di design (violazione di più principi
SOLID):
class GestioneOrdine {
    public void ProcessaOrdine(Ordine o) {
        Valida(o);
        SalvaSuDatabase(o);
        InviaEmailConferma(o);
    }
    private void Valida(Ordine o) { /* valida dati ordine (logica di
validazione) */ }
    private void SalvaSuDatabase(Ordine o)
{ /* salva ordine su DB (logica di persistenza) */ }
    private void InviaEmailConferma(Ordine o) { /* invia email al cliente
(logica di notifica) */ }
}
// Questa classe fa troppe cose: valida, salva e notifica – violando SRP e
rendendo difficile estendere/modificare singole parti.
```

Slide 13: Single Responsibility Principle (SRP) – Problema

Il **Single Responsibility Principle (SRP)** afferma che *una classe dovrebbe avere una ed una sola responsabilità*, cioè un solo motivo per poter cambiare. Una violazione comune è creare classi "Dios" o gestori generici che accumulano funzioni disparate. Quando una classe ha più di una responsabilità,

diventa più difficile **manutenere** o modificare il codice: un cambiamento per una funzionalità può impattare inaspettatamente anche l'altra. Inoltre, testare unitariamente una classe con troppe responsabilità è complicato.

Nel nostro esempio precedente, la classe `GestioneOrdine` viola SRP: gestisce sia la validazione, sia il salvataggio su database, sia l'invio di email. Ha quindi più motivi indipendenti per cambiare (cambi nelle regole di validazione, modifica del database o del formato email). Questa mancanza di **coesione** rende la classe fragile e difficile da estendere.

```
// Esempio di violazione SRP: classe con più responsabilità
class GestioneOrdine {
    public void ProcessaOrdine(Ordine o) {
        Valida(o);
        SalvaSuDB(o);
        InviaConfermaEmail(o);
    }
    private void Valida(Ordine o) { /* verifica dati ordine */ }
    private void SalvaSuDB(Ordine o) { /* persiste ordine nel database */ }
    private void InviaConfermaEmail(Ordine o) { /* invia email di conferma
*/ }
}
// Problema: GestioneOrdine fa validazione, persistenza e notifica - tre
responsabilità in una sola classe (SRP violato).
```

Slide 14: Single Responsibility Principle (SRP) – Soluzione

Per rispettare SRP, suddividiamo le responsabilità di `GestioneOrdine` in classi diverse, ognuna focalizzata su un compito. Otterremo classi più **semplici** e isolate: ad esempio `OrderValidator` per la validazione, `OrderRepository` per la persistenza e `NotificatoreEmail` per l'invio di conferme. La classe principale `OrderProcessor` coordina il flusso ma delega i dettagli alle componenti specializzate. Così ogni classe ha **un solo motivo di modifica** (ad es. cambiare la logica di validazione non tocca le altre).

Questo porta a un design con **alta coesione** (ogni classe fa bene una cosa) e basso accoppiamento (le classi interagiscono tramite astrazioni come interfacce). È più facile modificare o sostituire una parte senza impattare il resto, e testare ciascuna classe individualmente (ad esempio, testare il `OrderValidator` isolatamente dal database). Ecco come potrebbe apparire la soluzione:

```
// Interfacce per responsabilità separate:
interface IOrderValidator { void Valida(Ordine o); }
interface IOrderRepository { void Salva(Ordine o); }
interface INotificatore { void InviaConferma(Ordine o); }

// Implementazioni concrete (possibili):
class OrderValidator : IOrderValidator {
    public void Valida(Ordine o) { /* logica di validazione ordine */ }
}
class OrderRepository : IOrderRepository {
    public void Salva(Ordine o) { /* salva ordine su DB */ }
```

```
}
class NotificatoreEmail : INotificatore {
    public void InviaConferma(Ordine o) { /* invia email conferma */ }
}

// Classe che ha ora singola responsabilità di orchestrare usando le
dipendenze:
class OrderProcessor {
    private IOrderValidator _val;
    private IOrderRepository _repo;
    private INotificatore _notif;
    public OrderProcessor(IOrderValidator v, IOrderRepository r,
INotificatore n) {
        _val = v; _repo = r; _notif = n;
    }
    public void ProcessaOrdine(Ordine o) {
        _val.Valida(o);
        _repo.Salva(o);
        _notif.InviaConferma(o);
    }
}
// OrderProcessor delega compiti specifici a classi specializzate,
rispettando SRP.
```

Slide 15: Open/Closed Principle (OCP) – Problema

L'**Open/Closed Principle (OCP)** stabilisce che *il software dovrebbe essere aperto all'estensione, ma chiuso alla modifica*. In pratica, dovremmo poter aggiungere nuove funzionalità senza modificare il codice esistente, per evitare di introdurre bug nei componenti già testati. Una violazione tipica sono funzioni o classi che usano lunghi **switch/cascade if** per differenziare comportamenti in base a un tipo: aggiungere un nuovo tipo richiede di modificare quella funzione esistente.

Ad esempio, consideriamo una funzione che calcola l'area di diverse forme geometriche verificando il tipo dell'oggetto. Se vogliamo supportare un nuovo tipo di forma, dovremo **modificare** questa funzione aggiungendo un nuovo ramo condizionale, violando OCP. Il codice di seguito illustra questo scenario: l'aggiunta di un nuovo poligono richiederà di editare la logica esistente, con rischio di errori e impatto sulle altre parti.

```
// Violazione di OCP: funzione che deve essere modificata per estendere nuove
forme
double CalcolaArea(object forma) {
    if (forma is Cerchio) {
        Cerchio c = (Cerchio)forma;
        return Math.PI * c.Raggio * c.Raggio;
    } else if (forma is Rettangolo) {
        Rettangolo r = (Rettangolo)forma;
        return r.Larghezza * r.Altezza;
    } else {
        throw new ArgumentException("Tipo di forma non supportato");
    }
```

```
    // Nota: per una nuova forma (es. Triangolo) dovremmo aggiungere un altro
else if.
}
```

Slide 16: Open/Closed Principle (OCP) – Soluzione

Per applicare OCP, utilizziamo il **polimorfismo**: definendo un'interfaccia o classe base comune, ogni nuovo tipo concreto implementa il proprio comportamento, così il codice esistente non necessita modifiche. Nel caso delle forme geometriche, possiamo introdurre una classe astratta `Forma` con un metodo astratto `Area()`. Ogni sottoclasse (`Cerchio`, `Rettangolo`, ecc.) override il metodo per calcolare la propria area. La logica di calcolo è incapsulata nelle classi specifiche, e il codice che usa `Forma` può chiamare `Area()` senza conoscere i dettagli.

In questo design, il sistema è **aperto all'estensione** (possiamo aggiungere nuove forme creando nuove sottoclassi di `Forma`) ma **chiuso alla modifica** del codice esistente (la funzione che usa `Area()` rimane invariata). Ciò riduce i rischi di regressioni e segue anche il principio "Programmare per interfacce, non per implementazioni". Ecco l'esempio corretto:

```
abstract class Forma {
    public abstract double Area();
}

class Cerchio : Forma {
    public double Raggio { get; }
    public Cerchio(double r) { Raggio = r; }
    public override double Area() => Math.PI * Raggio * Raggio;
}

class Rettangolo : Forma {
    public double Larghezza { get; }
    public double Altezza { get; }
    public Rettangolo(double larg, double alt) { Larghezza = larg; Altezza =
alt; }
    public override double Area() => Larghezza * Altezza;
}

// Utilizzo polimorfo:
Forma f1 = new Cerchio(5);
Forma f2 = new Rettangolo(4, 3);
Console.WriteLine(f1.Area());  // 78.5..., calcola area cerchio
Console.WriteLine(f2.Area());  // 12, calcola area rettangolo
// Aggiungere una nuova forma (es. Triangolo) implica creare una nuova classe
Triangolo : Forma, senza modificare il codice esistente.
```

Slide 17: Liskov Substitution Principle (LSP) – Problema

Il **Liskov Substitution Principle (LSP)** afferma che *le classi derivate dovrebbero poter essere sostituite alle loro classi base senza alterare la correttezza del programma*. In altre parole, ovunque sia previsto un oggetto di tipo base, possiamo usare un oggetto di tipo derivato e il comportamento atteso non

cambia. La violazione di LSP accade quando una sottoclasse **non preserva il contratto** della superclasse, ad esempio restringendo le condizioni o l'ambito di validità dei metodi base.

Un esempio classico: supponiamo una classe base `Uccello` con un metodo `Vola()`. Se creiamo una sottoclasse `Pinguino` che eredita da `Uccello` ma non può volare, e magari implementa `Vola()` lanciando un'eccezione, avremo violato LSP. Un codice che itera su una lista di `Uccello` chiamando `Vola()` funzionerebbe per la maggior parte degli uccelli, ma si romperebbe incontrando un `Pinguino`. Ciò introduce un **comportamento inatteso** e rompe la sostituibilità.

```
class Uccello {
    public virtual void Vola() {
        Console.WriteLine("Sto volando!");
    }
}

class Pinguino : Uccello {
    public override void Vola() {
        throw new InvalidOperationException("Il pinguino non può volare!");
    }
}

// Codice che viola LSP:
Uccello u = new Pinguino();
u.Vola();  // Ci si aspetterebbe che voli (come definito in Uccello), invece
lancia eccezione a runtime.
```

Slide 18: Liskov Substitution Principle (LSP) – Soluzione

Per rispettare LSP, bisogna progettare la gerarchia di classi in modo che le sottoclassi **mantengano le promesse** fatte dalla classe base. Nel caso precedente, l'errore è aver forzato `Pinguino` a essere un `Uccello` generico dotato di `Vola()`. Una soluzione è rimuovere dalla superclasse qualsiasi assunzione che *tutti* i sottotipi possano volare. Possiamo creare un'interfaccia separata `IVolante` implementata solo dagli uccelli che volano, oppure introdurre nella classe base un comportamento neutro (es. `Vola()` vuoto) che i pinguini possono override senza effetti collaterali.

Un approccio migliore è modellare l'ereditarietà secondo la realtà: far sì che `Pinguino` non abbia un metodo volare, mentre classi come `Aquila` implementino un'interfaccia volabile. Così chi usa `Uccello` + `IVolante` potrà chiamare `Vola()` solo su quelli che possono farlo. In generale, ogni precondizione aggiuntiva o postcondizione violata nella sottoclasse rappresenta una violazione LSP. Progettare con astrazioni appropriate evita questi problemi.

```
interface IVolante { void Vola(); }

class Pinguino /* : Uccello, not implementing IVolante */ {
    public void Nuota() { Console.WriteLine("Il pinguino nuota."); }
}

class Aquila : IVolante {
```

```
    public void Vola() { Console.WriteLine("L'aquila vola in alto nel
cielo."); }
}

// Utilizzo corretto:
List<IVolante> volatili = new List<IVolante> { new Aquila() /* , ... altri
animali volanti */ };
foreach (var volatileObj in volatili) {
    volatileObj.Vola();  // Ogni oggetto in questa lista può volare, LSP
preservato.
}
// Pinguino non è incluso nella lista dei "IVolante", quindi non rompe
l'aspettativa di volo.
```

Slide 19: Interface Segregation Principle (ISP) – Problema

L'**Interface Segregation Principle (ISP)** afferma che *è preferibile avere molteplici interfacce specifiche piuttosto che una sola interfaccia generale e "grassa"*. I client non dovrebbero essere costretti a dipendere da metodi che non utilizzano. Una violazione tipica è la definizione di un'unica interfaccia che cumula troppe operazioni disparate: le classi che la implementano potrebbero dover fornire implementazioni vuote o fittizie per metodi irrilevanti per loro.

Ad esempio, immaginiamo un'interfaccia `IMultifunzione` che includa metodi per **stampare**, **scansionare** e **inviare fax**. Un dispositivo come una stampante economica che non ha funzionalità fax si troverebbe costretto a implementare `InviaFax` magari lanciando eccezioni o lasciandolo vuoto. Questo design è poco pulito: i cambiamenti a `IMultifunzione` impattano tutte le implementazioni, anche dove non servono. Inoltre viola la regola di avere interfacce piccole e mirate.

```
// Interfaccia "grassa" (viola ISP):
interface IMultifunzione {
    void StampaDocumento(string doc);
    void ScansionaDocumento(string doc);
    void InviaFax(string doc);
}

class StampanteEconomica : IMultifunzione {
    public void StampaDocumento(string doc) => Console.WriteLine($"Stampo:
{doc}");
    public void ScansionaDocumento(string doc) =>
Console.WriteLine($"Scansiono: {doc}");
    public void InviaFax(string doc) {
        // Questa stampante non supporta il fax
        throw new NotSupportedException("Fax non supportato da
StampanteEconomica");
    }
}
// StampanteEconomica è costretta ad avere un metodo di fax che non può
implementare realmente (ISP violato).
```

Slide 20: Interface Segregation Principle (ISP) – Soluzione

La soluzione è suddividere l'interfaccia ampia in **interfacce più piccole e mirate** ai diversi gruppi di funzionalità. Nel nostro esempio, potremmo creare interfacce separate: `IStampante` con il metodo `StampaDocumento`, `IScanner` con `ScansionaDocumento` e `IFax` con `InviaFax`. Così una classe implementa solo ciò che le serve: `StampanteEconomica` implementerà solo `IStampante` (e magari `IScanner` se scansiona) ma **non dipenderà** da `IFax`. In questo modo le modifiche a un'interfaccia (es. aggiungere un metodo fax) non toccano le classi che non hanno quella capacità.

Questo approccio aumenta la coesione delle interfacce e riduce l'accoppiamento tra componenti non correlati. Ogni classe "vede" soltanto i metodi di cui ha bisogno. Risulta anche più chiaro quali funzionalità supporta una classe (basta guardare quali interfacce implementa). L'ISP, in sintesi, porta a API più pulite e modulari.

```
interface IStampante { void StampaDocumento(string doc); }
interface IScanner { void ScansionaDocumento(string doc); }
interface IFax { void InviaFax(string doc); }

// Le classi implementano solo le interfacce necessarie:
class StampanteEconomica : IStampante, IScanner {
    public void StampaDocumento(string doc) => Console.WriteLine($"Stampo:
{doc}");
    public void ScansionaDocumento(string doc) =>
Console.WriteLine($"Scansiono: {doc}");
    // Nota: nessuna implementazione di IFax, perché questo modello non
supporta fax.
}

class StampanteAvanzata : IStampante, IScanner, IFax {
    public void StampaDocumento(string doc) { /* stampa */ }
    public void ScansionaDocumento(string doc) { /* scansione */ }
    public void InviaFax(string doc) { /* invio fax */ }
}
// Ora ciascuna classe ha solo i metodi pertinenti. ISP rispettato, niente
metodi inutilizzati o non implementabili.
```

Slide 21: Dependency Inversion Principle (DIP) – Problema

Il **Dependency Inversion Principle (DIP)** stabilisce che *le classi alto livello non dovrebbero dipendere da classi di basso livello; entrambi dovrebbero dipendere da astrazioni*. Inoltre, le astrazioni non dovrebbero dipendere dai dettagli, ma i dettagli dalle astrazioni. In pratica, questo principio incoraggia l'uso di **interfacce** o classi astratte per invertire la dipendenza tradizionale. Una violazione DIP comune è quando una classe di alto livello istanzia direttamente (dipende da) una classe concreta di basso livello, rendendo il codice fortemente accoppiato.

Ad esempio, una classe `DataExporter` che salva dati su file potrebbe creare internamente un oggetto `FileWriter`. Così `DataExporter` dipende dalla implementazione concreta del salvataggio (dettaglio). Se volessimo esportare i dati su database invece che su file, dovremmo modificare `DataExporter`. Questo disegno è rigido e poco testabile (non possiamo sostituire facilmente

`FileWriter` con un finto oggetto nei test). Il DIP qui è violato perché il modulo di alto livello conosce troppo dei dettagli di basso livello.

```
// Violazione DIP: dipendenza diretta da classe concreta
class DataExporter {
    public void EsportaDati(List<string> dati) {
        var fileWriter = new FileWriter();   // Dipendenza concreta creata
internamente
        fileWriter.SalvaSuFile(dati);
    }
}

class FileWriter {
    public void SalvaSuFile(List<string> dati) {
        // Logica per salvare i dati su un file
        Console.WriteLine("Dati salvati su file.");
    }
}
// DataExporter è legata a FileWriter. Per cambiare destinazione (es. DB),
bisogna modificare DataExporter.
```

Slide 22: Dependency Inversion Principle (DIP) – Soluzione

Per rispettare DIP, introduciamo un'**astrazione** per la funzionalità di salvataggio dati, ad esempio un'interfaccia `ISalvataggio` con un metodo `Salva(List<string>)`. `DataExporter` dipenderà da `ISalvataggio` (non sa nulla di come avviene il salvataggio), e potremo avere implementazioni concrete come `FileWriter` o `DatabaseWriter` che realizzano `ISalvataggio`. Ora `DataExporter` è un modulo di alto livello **indipendente dai dettagli** concreti: per cambiare la modalità di output basta fornirgli un oggetto diverso che implementa `ISalvataggio`, senza toccare il codice di DataExporter.

Questa inversione delle dipendenze semplifica anche il testing: possiamo passare a `DataExporter` una finta implementazione di `ISalvataggio` nei test unitari (ad esempio una classe `FakeSalvataggio` che scrive su una lista in memoria) per verificare la logica senza toccare filesystem o database. Il DIP spesso va di pari passo con l'**Dependency Injection**, dove un contenitore esterno fornisce l'implementazione concreta appropriata all'interfaccia richiesta.

```
// Definizione dell'astrazione:
interface ISalvataggio {
    void Salva(List<string> dati);
}

// Implementazioni concrete (dettagli) che dipendono dall'interfaccia:
class FileWriter : ISalvataggio {
    public void Salva(List<string> dati) {
        Console.WriteLine("Salvataggio su file completato.");
        // ... (codice per scrivere i dati su file)
    }
}
```

```
class DatabaseWriter : ISalvataggio {
    public void Salva(List<string> dati) {
        Console.WriteLine("Salvataggio su database completato.");
        // ... (codice per inserire dati nel DB)
    }
}

// Modulo di alto livello che dipende dall'astrazione ISalvataggio:
class DataExporter {
    private ISalvataggio _salvataggio;
    public DataExporter(ISalvataggio salvataggio) {
        _salvataggio = salvataggio;
    }
    public void EsportaDati(List<string> dati) {
        _salvataggio.Salva(dati);
    }
}

// Utilizzo:
DataExporter exporter = new DataExporter(new FileWriter());
exporter.EsportaDati(new List<string>{"dato1","dato2"});
// Possiamo sostituire FileWriter con DatabaseWriter senza cambiare
DataExporter.
```

Inversione di Controllo (IoC)

Slide 23: Inversione di Controllo (IoC) – Introduzione

L'**Inversione di Controllo (IoC)** è un principio architetturale secondo cui un componente di livello superiore non si occupa direttamente di gestire il **flusso di controllo** o di creare le proprie dipendenze, ma delega questo compito a una parte esterna (tipicamente un framework o un container). In altre parole, si inverte la responsabilità del controllo: invece di avere il codice dell'applicazione che chiama le librerie, è un **framework esterno** che chiama il codice applicativo fornito. Questo approccio è anche noto come principio di Hollywood: "*Don't call us, we'll call you*".

Il beneficio di IoC è un codice più **decoupled** e modulare. L'applicazione definisce cosa fare (logica di business) e dichiara le dipendenze di cui ha bisogno, ma è il contenitore/framework a fornire tali dipendenze e a orchestrare le chiamate. Esempi comuni di IoC sono i framework GUI o web: ad esempio, in ASP.NET il runtime chiama i metodi dei controller anziché il contrario. IoC crea punti di estensibilità (ad es. registrando callback, plugin, handler di eventi) dove il nostro codice viene eseguito dal framework al verificarsi di certe condizioni.

```
// Differenza tra approccio tradizionale e IoC:

// Approccio tradizionale (app chiama libreria):
App avvia -> Libreria.DoSomething();  // L'applicazione controlla il flusso,
chiama libreria

// Inversione di Controllo (framework chiama app):
```

```
Framework framework = new Framework();
framework.RegistraCallback(funzioneUtente);
framework.Esegui();
// In questo caso, il framework durante la sua esecuzione chiamerà
funzioneUtente() al momento opportuno,
// invertendo il flusso: ora è il framework a comandare, e l'app fornisce
solo il codice da eseguire.
```

Slide 24: Inversione di Controllo – Esempio con Eventi

Un esempio pratico di IoC in C# è il meccanismo degli **eventi**. Quando un oggetto espone un evento, il codice utente può **registrare** un proprio metodo (delegate) da eseguire quando l'evento si verifica. Qui il controllo è invertito: è l'oggetto che possiede l'evento (ad es. un pulsante) a chiamare il metodo dell'utente quando, poniamo, viene cliccato, invece che l'utente controllare attivamente lo stato del pulsante. Questo elimina la necessità per il codice utente di interrogare continuamente l'oggetto (polling) e decoupla il *quando* (deciso internamente al framework) dal *cosa fare* (fornito dall'utente).

Nell'esempio seguente, la classe `Bottone` ha un evento `Cliccato`. Il codice esterno (`InterfacciaUtente`) si registra all'evento passando una lambda (funzione anonima) che sarà eseguita automaticamente dal bottone (ad esempio all'interno di `Click()`). Questo mostra chiaramente l'IoC: **il bottone chiama il codice esterno** quando serve, non il contrario.

```
class Bottone {
    public event Action Cliccato;  // Event handler (delegati Action senza
parametri)

    public void Click() {
        Console.WriteLine("Bottone premuto, eseguo gli handler...");
        Cliccato?.Invoke();  // Chiama tutti i metodi registrati all'evento
    }
}

class InterfacciaUtente {
    public InterfacciaUtente() {
        Bottone btn = new Bottone();
        // Registrazione di un callback (lambda) all'evento Cliccato:
        btn.Cliccato += () => Console.WriteLine("Evento bottone Cliccato
ricevuto! Aggiorno UI.");
        // Simulazione di un click sul bottone:
        btn.Click();
    }
}
// Output simulato:
// Bottone premuto, eseguo gli handler...
// Evento bottone Cliccato ricevuto! Aggiorno UI.
```

Slide 25: Inversione di Controllo – Contenitore IoC

Un altro aspetto molto diffuso dell'IoC è l'uso di **contenitori IoC** (o container DI), che gestiscono la creazione e fornitura di oggetti. Invece di istanziare manualmente le dipendenze con `new`, si registra nel contenitore come creare determinati tipi e poi si **richiede** al contenitore di fornire le istanze necessarie. Il container dunque controlla la costruzione degli oggetti (inversione del controllo di creazione) e può gestire anche cicli di vita (singleton, transient, etc.).

Ad esempio, in un'app potremmo configurare un contenitore registrando che per l'interfaccia `INotificatore` la classe concreta da usare è `EmailNotificatore`. Quando il programma avrà bisogno di un `INotificatore`, chiederà al container un'istanza: sarà il container a decidere **quale classe concreta** fornire e a crearla. Ciò rende semplice cambiare implementazione (basta cambiare la registrazione, non il codice ovunque). Ecco un esempio semplificato:

```
// Pseudo-codice per un semplice contenitore IoC:
var container = new SimpleContainer();
container.Registra<INotificatore,
EmailNotificatore>();   // Registra che per INotificatore usare
EmailNotificatore
container.Registra<ILogger, ConsoleLogger>();             // Per ILogger
usare ConsoleLogger

// ... altrove nel codice, quando servono le dipendenze:
INotificatore notif = container.Risolve<INotificatore>();  // Il container
crea un EmailNotificatore
ILogger log = container.Risolve<ILogger>();               // Il container
crea un ConsoleLogger

notif.Invia("Messaggio di test");
log.Log("Notifica inviata con successo.");
// Qui il codice non ha usato 'new': ha delegato al contenitore la fornitura
degli oggetti (Inversione del Controllo di istanziazione).
```

Slide 26: Inversione di Controllo vs Dependency Injection

Inversione di Controllo e **Dependency Injection** sono strettamente correlati ma non identici. IoC è un principio generale: riguarda l'inversione del flusso di chiamate o di gestione rispetto al metodo tradizionale. **Dependency Injection (DI)** è una forma specifica di IoC focalizzata sulle dipendenze: invece di far sì che un oggetto crei le proprie dipendenze, esse gli vengono **iniettate dall'esterno**, tipicamente da un container IoC. In altre parole, DI è un pattern che realizza IoC nel contesto della gestione delle dipendenze tra oggetti.

Un'altra forma di IoC, ad esempio, è il *Service Locator* (un oggetto globale da cui i componenti chiedono le dipendenze) o il meccanismo di callback/eventi come visto. La differenza chiave: IoC parla di *chi* chiama *chi*, mentre DI parla di *come* un oggetto ottiene le sue dipendenze. Spesso in un framework moderno (come ASP.NET Core) troviamo entrambi: il framework controlla il ciclo di vita delle componenti (IoC) e fornisce le dipendenze ai componenti (DI).

```
// Senza IoC/DI: creazione manuale delle dipendenze
var controller = new ReportController();
controller.Service = new ReportService();  // dipendenza assegnata
manualmente dal codice applicativo

// Con IoC + DI: è il container a creare e iniettare la dipendenza
ReportController controller = container.Risolve<ReportController>();
// Il contenitore istanzia ReportController e automaticamente gli passa un
ReportService (rispettando le dipendenze dichiarate).
```

Slide 27: Vantaggi di IoC

L'Inversione di Controllo offre diversi vantaggi progettuali. Innanzitutto, aumenta il **disaccoppiamento**: i componenti dell'applicazione non conoscono i dettagli della creazione o del flusso globale, si concentrano sulla logica di business. Ciò facilita la **manutenibilità** e l'estendibilità, perché cambiamenti nell'implementazione (ad es. sostituire un componente con un altro) non richiedono di modificare il codice ovunque, ma solo la configurazione o implementazione specifica.

Un altro grande beneficio è la **testabilità**. In un contesto IoC+DI, possiamo facilmente sostituire dipendenze reali con mock o stub nei test unitari, isolando il comportamento da verificare. Inoltre, delegando a un framework, si riducono i boilerplate: ad esempio, non serve scrivere codice ripetitivo per passare oggetti in giro, lo fa il container. L'IoC incoraggia un design a componenti plug-and-play: moduli intercambiabili e sviluppo parallelo (diversi team possono sviluppare componenti purché rispettino le interfacce).

```
// Esempio: IoC semplifica il testing permettendo injection di mock
public class ReportController {
    private IReportService _service;
    public ReportController(IReportService service) { _service = service; }

    public string GeneraReport(int id) {
        return _service.CreaReport(id);
    }
}

// In produzione, il container inietterà l'implementazione reale di
IReportService:
IReportService realService = new ReportServiceDB();
var controller = new ReportController(realService);

// Nei test, possiamo iniettare un fake o mock:
class FakeReportService : IReportService {
    public string CreaReport(int id) { return "report finto"; }
}
var testController = new ReportController(new FakeReportService());
Console.WriteLine(testController.GeneraReport(5));  // Usa il servizio finto,
isolando il test dall'accesso DB
```

Slide 28: IoC nei Framework (es. ASP.NET Core)

Molti framework moderni adottano IoC come architettura di base. Prendiamo ad esempio **ASP.NET Core**: il framework gestisce il ciclo di vita dell'applicazione, crea gli oggetti controller, li **istanzia** e li popola con le dipendenze necessarie. Il nostro ruolo da sviluppatori è definire controller, servizi e configurarli nel container DI (ad es. nel `Program.cs` o `Startup.cs`). Sarà poi ASP.NET Core a chiamare i metodi dei controller quando arrivano richieste HTTP (IoC) e a **iniettare automaticamente** le dipendenze nel costruttore del controller (DI).

Per esempio, definendo un controller che richiede nel costruttore un servizio `IRepository`, non istanziamo nulla manualmente: ASP.NET Core vede la richiesta di `IRepository`, cerca nel contenitore la registrazione e gli passa un'istanza appropriata. Questo rende il codice del controller pulito e focalizzato solo sulla logica HTTP/Business, delegando al framework tutta la gestione di infrastruttura.

```
// Esempio semplificato di un controller ASP.NET Core con DI:
public class OrdiniController : Controller {
    private readonly IOrdiniRepository _repo;

    // Il framework inietterà IOrdiniRepository qui se è stato registrato nel
container
    public OrdiniController(IOrdiniRepository repo) {
        _repo = repo;
    }

    [HttpGet("/ordini/{id}")]
    public IActionResult GetOrdine(int id) {
        var ordine = _repo.TrovaPerId(id);
        if (ordine == null) return NotFound();
        return Ok(ordine);
    }
}

// Program.cs - configurazione del container DI in ASP.NET Core:
builder.Services.AddScoped<IOrdiniRepository, OrdiniRepositorySQL>();
// Quando ASP.NET creerà OrdiniController, fornirà un OrdiniRepositorySQL
come IOrdiniRepository automaticamente.
```

Slide 29: Implementare IoC manualmente – Factories

Anche senza un framework o container sofisticato, possiamo applicare IoC/DI manualmente. Ad esempio, usando pattern creazionali come Factory o Abstract Factory, centralizziamo la creazione di oggetti in un punto. Il codice di alto livello chiederà all'oggetto factory l'istanza necessaria invece di creare direttamente (questo è già un'inversione del controllo di creazione). In fase di test, la factory può restituire implementazioni alternative o stub.

Un altro approccio manuale è passare le dipendenze tramite costruttore (constructor injection) o proprietà (setter injection) come visto nella sezione DI. Ad esempio, per testare o cambiare implementazione, possiamo costruire manualmente l'oggetto con le sue dipendenze desiderate. Il concetto IoC chiave qui è che la **configurazione dell'applicazione** (composizione degli oggetti) avviene in uno strato esterno, separato dalla logica interna delle classi.

```
// Esempio: configurazione manuale delle dipendenze (senza container)
IEmailService emailSvc = new
EmailService();        // istanziamo manualmente la dipendenza
IUserRepository userRepo = new UserRepositoryDB();  // un'altra dipendenza
UserController controller = new UserController(emailSvc, userRepo);
// Abbiamo invertito il controllo: invece che UserController crei da sé
EmailService e UserRepository,
// li stiamo passando dall'esterno. Possiamo facilmente sostituire emailSvc o
userRepo qui se servisse.

// Nel test potremmo fare:
IEmailService fakeEmail = new FakeEmailService();
IUserRepository fakeRepo = new FakeUserRepo();
UserController testController = new UserController(fakeEmail, fakeRepo);
// Così testController usa implementazioni finte, dimostrando la flessibilità
data dall'IoC manuale.
```

Slide 30: Service Locator (e considerazioni)

Il **Service Locator** è un altro pattern per ottenere inversione di controllo. Si tratta di un oggetto "registratore centrale" presso cui si registrano i servizi e dal quale i componenti possono richiedere le loro dipendenze. Invece di ricevere le dipendenze nel costruttore (DI), una classe potrebbe ottenere ciò che le serve chiamando il locator. Questo fornisce IoC perché la classe non conosce la concreta implementazione (il locator gliela dà), ma al costo di dipendere comunque dallo locator stesso.

Il Service Locator è considerato da alcuni un **anti-pattern** perché tende a nascondere le dipendenze (guardando la classe non è chiaro quali servizi usi, li pesca dentro il locator globale) e può incoraggiare un design meno chiaro. Inoltre sposta l'accoppiamento a livello del locator globale. Tuttavia, può essere utile in alcuni contesti legacy o dove la DI esplicita non è praticabile. In generale, è preferibile usare DI, ma vale la pena conoscere il locator come variante.

```
// Esempio semplificato di Service Locator:
static class ServiceLocator {
    private static Dictionary<Type, object> _servizi = new Dictionary<Type,
object>();

    public static void Registra<T>(T servizio) {
        _servizi[typeof(T)] = servizio;
    }

    public static T Ottieni<T>() {
        return (T)_servizi[typeof(T)];
    }
}

// Registrazione dei servizi (di solito all'avvio dell'app):
ServiceLocator.Registra<INotificatore>(new EmailNotificatore());
ServiceLocator.Registra<ILogger>(new ConsoleLogger());
```

```
// Dentro una classe qualsiasi, uso del locator (invece di DI esplicita):
class ResocontoService {
    public void GeneraResoconto() {
        var logger = ServiceLocator.Ottieni<ILogger>();
        logger.Log("Inizio generazione resoconto");
        // ... logica ...
        logger.Log("Resoconto generato");
    }
}
// Nota: ResocontoService dipende implicitamente da ILogger, ma ciò non è
evidente dalla firma della classe (dipendenza nascosta).
```

Slide 31: Quando evitare IoC?

Sebbene IoC e DI apportino molti benefici in sistemi complessi, **non sempre** sono necessari o vantaggiosi. In applicazioni piccole o script, introdurre un container IoC o una miriade di interfacce può aggiungere **complessità** inutile. Se il numero di dipendenze è limitato e il cambiamento previsto è minimo, il codice potrebbe essere più chiaro gestendo direttamente le istanze senza layer aggiuntivi. È importante valutare il rapporto costo/beneficio: IoC introduce un livello di indirezione che richiede familiarità da parte del team e può rendere il debug leggermente più ostico.

Un altro caso in cui stare attenti è quando un eccesso di astrazioni complica la struttura: ad esempio, creare interfacce per ogni cosa anche se si avranno sempre e solo una o due implementazioni può essere over-engineering. Inoltre, l'abuso di Service Locator globale può riportare di fatto al problema delle variabili globali. **Semplicità prima di tutto**: applicare IoC dove serve modularità, testabilità e scalabilità; evitarlo quando introduce più problemi di quanti ne risolve in un dato contesto.

```
// Esempio: in un programma semplice, creare oggetti direttamente può essere
più immediato
class Stampante {
    public void Stampa(string msg) => Console.WriteLine(msg);
}

class Program {
    static void Main() {
        Stampante stampante = new Stampante();  // istanziazione diretta,
chiara e semplice
        stampante.Stampa("Hello World");
    }
}
// In un caso così elementare, introdurre un'interfaccia IStampante, un
container IoC, ecc., sarebbe eccessivo.
```

Slide 32: IoC e Test Unitari

Come accennato, l'IoC brilla particolarmente nel contesto dei **test automatizzati**. Grazie alla Dependency Injection (forma di IoC), possiamo fornire ai nostri oggetti delle dipendenze fittizie (mock, stub o fake) al posto di quelle reali, per isolare il comportamento durante i test unitari. Ad esempio, se una classe A usa un servizio B esterno (DB, API, etc.), iniettando un fake di B in A possiamo testare A

senza coinvolgere davvero B (evitando chiamate lente o non deterministiche). Questo permette test più veloci, isolati e ripetibili.

Consideriamo un servizio `UserService` che dipende da un database (interfaccia `IDatabase`). Senza IoC/DI, `UserService` magari istanzia direttamente `DatabaseSql` al suo interno, rendendo difficile sostituirlo. Con IoC, `UserService` riceve `IDatabase` nel costruttore: in produzione gli passeremo `DatabaseSql`, mentre nel test gli passeremo un `FakeDatabase` che simula il comportamento (ad esempio, restituisce sempre gli stessi dati predefiniti). Il risultato è che testare `UserService` non richiede un vero DB e possiamo verificare la logica in modo affidabile.

```
// Servizio da testare, dipende da un database (astratto):
public interface IDatabase {
    string GetEmailUtente(int userId);
}
public class UserService {
    private IDatabase _db;
    public UserService(IDatabase db) {
        _db = db;
    }
    public string RecuperaEmail(int userId) {
        // Logica (da testare) magari con qualche elaborazione
        string email = _db.GetEmailUtente(userId);
        return email?.ToLower() ?? "email_non_trovata";
    }
}

// Implementazione reale (ad esempio, query su DB):
public class DatabaseSql : IDatabase {
    public string GetEmailUtente(int userId) {
        // ... chiamata SQL per ottenere l'email ...
        return "reale@example.com";
    }
}

// Implementazione fake per test:
public class FakeDatabase : IDatabase {
    public string GetEmailUtente(int userId) {
        return "fake@example.com";  // valore fisso o simulato per test
    }
}

// Test unitario:
var servizio = new UserService(new FakeDatabase());
string risultato = servizio.RecuperaEmail(42);
Console.WriteLine(risultato);  // "fake@example.com" (comportamento
prevedibile per testare la logica di UserService)
```

Slide 33: Riepilogo IoC

L'Inversione di Controllo è un principio chiave per progettare software estensibile e gestire la complessità delle dipendenze. **Riassumendo**: - Con IoC si **invertisce** il classico flusso: i componenti esterni (framework o container) orchestrano il ciclo di vita e le chiamate, mentre il nostro codice si inserisce sotto forma di plugin, callback, o viene costruito e chiamato dal container. - Per implementare IoC nelle dipendenze, usiamo il pattern di **Dependency Injection**: programmazione verso **astrazioni** (interfacce) e iniezione di implementazioni concrete dall'esterno. Ciò rende i moduli indipendenti dai dettagli concreti. - Vantaggi: codice modulare, testabile, **configurabile** (possiamo assemblare l'applicazione in modi diversi senza cambiare i moduli interni). Si evita l'**alto accoppiamento** e si facilita la manutenzione e l'aggiunta di funzionalità.

In pratica, per applicare IoC/DI:

```
/* Best practice IoC/DI:
   1. Progettare le classi perché dipendano da interfacce (astrazioni) invece
che da classi concrete.
   2. Utilizzare un contenitore DI o un modulo di configurazione centrale per
creare gli oggetti e iniettarvi le dipendenze necessarie.
   3. Lasciare che sia l'esterno (framework/container) a gestire il
"collaudo" tra oggetti – il codice applicativo non dovrebbe fare 'new' di
servizi core da cui dipende.
   4. Mantenere il numero di dipendenze di ciascuna classe ragionevole; se
una classe ha bisogno di troppe dipendenze, valutare se sta violando SRP
(forse va suddivisa).
*/
```

Dependency Injection (DI)

Slide 34: Dependency Injection (DI) – Introduzione

L'**Dependency Injection (iniezione delle dipendenze)** è un pattern di progettazione che realizza concretamente l'IoC per la gestione delle dipendenze tra oggetti. L'idea fondamentale è che un oggetto non crea direttamente le proprie dipendenze, ma le riceve già pronte dall'esterno (iniettate, tipicamente nel costruttore). In questo modo l'oggetto dipende da **astrazioni** (interfacce o classi base), mentre il compito di fornire un'implementazione concreta spetta a un assembler esterno (container DI o codice di configurazione).

Per esempio, se una classe `ReportService` necessita di un logger, invece di fare `logger = new ConsoleLogger()` al suo interno, dichiarerà nel costruttore un parametro `ILogger logger`. Sarà il chiamante a passare un `ConsoleLogger` o un altro logger concreto. Così `ReportService` non conosce i dettagli del logger (rispetta DIP). DI rende esplicite le dipendenze di una classe, agevolandone la comprensione e soprattutto permettendo la **sostituzione facile** delle implementazioni (utile per test, configurazioni diverse, ecc.).

```
// Classe con dependency injection via costruttore:
public class ReportService {
    private readonly ILogger _logger;
```

```
    public ReportService(ILogger logger) {
        _logger = logger;  // dipendenza iniettata nel costruttore
    }

    public void GeneraReport(string dati) {
        _logger.Log("Inizio generazione report");
        // ... logica di generazione ...
        _logger.Log("Report generato con successo");
    }
}

// Uso:
ILogger logger = new ConsoleLogger();
ReportService servizio = new ReportService(logger);  // iniettiamo la
dipendenza esternamente
servizio.GeneraReport("Analisi vendite 2025");
```

Slide 35: Constructor Injection

La **Constructor Injection** è la forma più comune e consigliata di DI. Le dipendenze di una classe vengono dichiarate come parametri nel suo **costruttore**, rendendo obbligatorio fornirle al momento della creazione dell'oggetto. Ciò garantisce che l'oggetto sia sempre in uno stato valido e completamente operativo (ha già tutto ciò che gli serve). Inoltre, rende molto chiaro quali sono le dipendenze di quella classe (basta guardare la firma del costruttore).

I vantaggi della constructor injection includono l'**immutabilità** delle dipendenze (possono essere `readonly` e non cambiare dopo l'istanziazione) e la possibilità di rilevare subito se manca una dipendenza (perché l'oggetto non può essere creato). Un potenziale svantaggio appare quando i parametri diventano molti: ciò potrebbe indicare che la classe sta facendo troppo (violazione SRP) o che può essere necessaria una factory per costruirla. In ogni caso, restare entro 3-4 dipendenze per costruttore di solito è gestibile.

```
// Esempio di constructor injection:
public interface IMailer { void InviaEmail(string destinatario, string
corpo); }

public class NotificaUtente {
    private readonly IMailer _mailer;
    private readonly ILogger _logger;

    // Entrambe le dipendenze vengono fornite tramite il costruttore
    public NotificaUtente(IMailer mailer, ILogger logger) {
        _mailer = mailer;
        _logger = logger;
    }

    public void InviaNotifica(string userEmail) {
        _logger.Log($"Invio notifica a {userEmail}");
        _mailer.InviaEmail(userEmail, "Hai un nuovo messaggio!");
        _logger.Log("Notifica inviata");
```

```
    }
}

// Quando creiamo NotificaUtente, dobbiamo passare implementazioni di IMailer
e ILogger:
IMailer mailer = new SmtpMailer();
ILogger logger = new FileLogger();
var notificatore = new NotificaUtente(mailer, logger);
// notificatore è pronto all'uso con le sue dipendenze assegnate.
```

Slide 36: Property Injection

La **Property Injection** (iniezione tramite proprietà) consiste nel fornire dipendenze attraverso proprietà pubbliche (o setter) dell'oggetto, piuttosto che attraverso il costruttore. Questo rende le dipendenze **opzionali** o configurabili dopo l'istanziazione iniziale. È utile quando alcune dipendenze non sono strettamente necessarie per il funzionamento base dell'oggetto, oppure quando esiste una dipendenza circolare che impedisce di passarla via costruttore senza complicazioni.

Con property injection, si crea l'oggetto con un costruttore vuoto o predefinito, e subito dopo si settano le dipendenze chiamando i setter. Bisogna fare attenzione a utilizzare questo metodo: se una dipendenza è fondamentale e dimentichiamo di impostarla, rischiamo di avere un oggetto in uno stato inconsistente. Spesso i framework DI usano property injection per particolari scenari, ma in generale preferiscono la constructor injection. Se usata, conviene documentare bene quali proprietà devono essere settate prima dell'uso.

```
public interface IValidator { bool Valida(string input); }

public class Configuratore {
    // Dipendenza opzionale tramite property:
    public IValidator? Validator { private get; set; }  // può non essere
impostato

    public void EseguiConfigurazione(string input) {
        if (Validator != null) {
            // Se un validator è disponibile, usalo:
            bool valido = Validator.Valida(input);
            if (!valido) {
                Console.WriteLine("Input di configurazione non valido.");
                return;
            }
        }
        Console.WriteLine("Configurazione eseguita con input: " + input);
    }
}

// Utilizzo con property injection:
var conf = new Configuratore();
// Iniettiamo facoltativamente un validatore:
conf.Validator = new SimpleValidator();
conf.EseguiConfigurazione("dato");
```

```
// Nota: Configuratore potrebbe funzionare anche senza Validator (la
proprietà rimane null), in tal caso salta la validazione.
```

Slide 37: Method Injection

La **Method Injection** prevede che la dipendenza venga passata come parametro in un metodo, solo nel momento in cui è necessaria. Questo pattern è meno comune, ma può essere utile quando una dipendenza serve esclusivamente per una specifica operazione e non ha senso mantenerla come stato interno dell'oggetto. Ad esempio, se occasionalmente abbiamo bisogno di un servizio per completare un'azione, possiamo prevedere un metodo `EseguiAzione(IServizio serv)` così che il chiamante fornisca il servizio al bisogno.

Questo mantiene la classe slegata da quella dipendenza per il resto del tempo. Lo svantaggio è che il metodo con injection deve essere chiamato con la dipendenza corretta, quindi c'è un pre-requisito d'uso (non sempre evidente) per quel metodo. In genere, method injection è usato in combinazione con altri tipi di DI, raramente come scelta primaria, a meno di casi molto specifici. Mantiene comunque il codice orientato alle **astrazioni** e permette injection di implementazioni diverse per chiamate diverse.

```
public interface ICrittografo {
    string Cifra(string testo);
}

public class Messenger {
    // Metodo che inietta la dipendenza solo per questa chiamata
    public void InviaMessaggio(string testo, ICrittografo crittografo) {
        string crittato = crittografo.Cifra(testo);
        Console.WriteLine($"Invio messaggio crittografato: {crittato}");
        // ... invio del messaggio crittato ...
    }
}

// Uso di method injection:
var messenger = new Messenger();
ICrittografo algoritmoA = new CrittografiaAES();
ICrittografo algoritmoB = new CrittografiaRSA();

// Possiamo scegliere l'algoritmo di crittografia per ogni invio:
messenger.InviaMessaggio("Salve", algoritmoA);
messenger.InviaMessaggio("Ciao", algoritmoB);
// Messenger non ha stato relativo al crittografo: a ogni chiamata possiamo
iniettare implementazioni diverse.
```

Slide 38: Dependency Injection – Contenitore DI

Nella pratica, la Dependency Injection è spesso supportata da un **contenitore DI** che automatizza la fornitura delle dipendenze. Il contenitore mantiene un registro di quali implementazioni usare per ciascuna interfaccia o classe e crea gli oggetti su richiesta, risolvendo ricorsivamente anche le dipendenze delle dipendenze. Ad esempio, il container di .NET

(Microsoft.Extensions.DependencyInjection) permette di configurare i servizi in fase di avvio e poi provvederà a iniettarli nei costruttori appropriati.

Per illustrare: supponiamo di avere un'interfaccia `IDataAccess` e una classe `DataAccessSql` concreta. Possiamo registrare nel container che a runtime ogni richiesta di `IDataAccess` deve essere soddisfatta con un'istanza di `DataAccessSql`. Quando chiederemo al container un oggetto che dipende da `IDataAccess` (es. un servizio business), il container creerà automaticamente un `DataAccessSql` e lo passerà. Ciò semplifica molto la **configurazione** dell'app ed evita boilerplate di creazione oggetti.

```
// Configurazione di un contenitore DI in .NET (esempio con Microsoft DI):
var services = new ServiceCollection();
services.AddTransient<ILogger, ConsoleLogger>();       // ogni volta che
serve ILogger -> ConsoleLogger
services.AddSingleton<IRepository, InMemoryRepo>();    // IRepository ->
InMemoryRepo come singleton
services.AddTransient<ReportService>();                // ReportService con
le sue dipendenze

using var provider = services.BuildServiceProvider();

// Risoluzione automatica delle dipendenze:
ReportService svc = provider.GetService<ReportService>()!;
// Il container ha istanziato ReportService, notato che richiede un
IRepository e un ILogger (nel suo costruttore),
// quindi ha creato un InMemoryRepo (singleton) e un ConsoleLogger (nuovo) e
li ha passati a ReportService.

// Ora possiamo usare svc sapendo che tutte le sue dipendenze sono state
soddisfatte:
svc.GeneraReportAnnuale();
```

Slide 39: Scope delle dipendenze (Transient, Singleton, Scoped)

I contenitori DI gestiscono anche il **ciclo di vita** (lifetime) delle istanze che creano. Le principali modalità sono: - **Transient**: ogni richiesta di una dipendenza produce una nuova istanza. Utile per servizi senza stato condiviso, che devono essere indipendenti ad ogni utilizzo. - **Singleton**: viene creata una sola istanza che viene riutilizzata per tutte le richieste future di quella dipendenza. Indicato per oggetti immutabili, thread-safe, o risorse costose da creare ma riutilizzabili (es. un HttpClient con configurazione). - **Scoped**: una via di mezzo, tipicamente in contesti web indica un'istanza unica per ogni *scope* operativo (ad esempio per ogni richiesta HTTP entrante un nuovo oggetto, condiviso all'interno di quella richiesta, ma diverso tra richieste differenti).

È importante scegliere lo scope giusto per evitare problemi: ad esempio, un servizio *singleton* che mantiene stato modficabile può causare errori se usato da più chiamanti contemporaneamente (thread safety). I transients evitano questo ma possono avere overhead se creazione/distruzione è pesante. Lo *scoped* è utile in context come web app dove vuoi condividere un oggetto (es. contesto DB) durante l'elaborazione di una singola richiesta, ma non oltre.

```
// Esempio di registrazione con diversi cicli di vita (usando .NET DI):
services.AddTransient<IOperation, Operation>();     // Transient: nuova
istanza per ogni risoluzione
services.AddScoped<IDbContext, AppDbContext>();     // Scoped: una istanza
per ciascuna request (in un'app web)
services.AddSingleton<ICache, MemoryCache>();       // Singleton: unica
istanza per tutta la durata dell'app

// Dimostrazione Transient vs Singleton:
using var provider = services.BuildServiceProvider();
var op1 = provider.GetService<IOperation>();
var op2 = provider.GetService<IOperation>();
Console.WriteLine(op1 != op2);  // True, transient produce istanze distinte

var cache1 = provider.GetService<ICache>();
var cache2 = provider.GetService<ICache>();
Console.WriteLine(object.ReferenceEquals(cache1, cache2));  // True,
singleton restituisce sempre la stessa istanza
```

Slide 40: Vantaggi della Dependency Injection

La Dependency Injection porta diversi benefici tangibili: - **Basso accoppiamento**: i componenti dipendono da interfacce astratte, quindi i dettagli concreti possono variare senza rompere il codice. Questo rende il sistema più resiliente ai cambiamenti (es: sostituire un database con un altro) e favorisce il riuso di componenti in contesti diversi. - **Testabilità**: come visto, grazie alla DI possiamo iniettare mock o stub facilmente, isolando i test. Le classi diventano più facili da testare perché non fanno nuove istanze hard-coded al loro interno. - **Flessibilità e configurazione**: è semplice configurare comportamenti diversi in base all'ambiente. Ad esempio, in produzione inietto un servizio reale, in sviluppo uno fake, in test uno mock, il tutto senza cambiare il codice delle classi, ma solo la configurazione del container o l'assembly delle dipendenze. - **Manutenibilità e leggibilità**: le dipendenze di una classe sono esplicite nel costruttore. Un nuovo sviluppatore vedendo il costruttore capisce subito di quali servizi quella classe ha bisogno, migliorando la **comprensione** del codice.

In sintesi, DI porta a un design pulito, modulare e orientato ai contratti (interfacce), che scala meglio in sistemi grandi.

```
// Scenario: la stessa logica di business con implementazioni diverse grazie
a DI
public class ImportazioneDati {
    private readonly IParser _parser;
    public ImportazioneDati(IParser parser) { _parser = parser; }

    public List<Record> Importa(string filePath) {
        string contenuto = File.ReadAllText(filePath);
        return _parser.Parsifica(contenuto);
    }
}

// Possiamo passare implementazioni diverse di IParser senza toccare
```

```
ImportazioneDati:
IParser csvParser = new CsvParser();
IParser jsonParser = new JsonParser();

var importCSV = new ImportazioneDati(csvParser);
var importJSON = new ImportazioneDati(jsonParser);
// importCSV e importJSON usano la stessa logica di ImportazioneDati ma con
parser differenti (flessibilità grazie a DI).
```

Slide 41: Svantaggi/Pitfalls della Dependency Injection

Pur avendo molti vantaggi, la DI presenta alcune **complessità** da gestire: - **Aumento di complessità iniziale**: per progetti piccoli o team inesperti, introdurre DI/IoC può essere eccessivo. C'è una curva di apprendimento per padroneggiare i contenitori DI e il debug può risultare meno intuitivo (poiché gli oggetti sono creati dietro le quinte). - **Indirezione**: seguire il flusso del programma può diventare più difficile. Per capire da dove proviene un oggetto, bisogna sapere cosa è registrato nel container. Se molti componenti sono iniettati, il percorso d'esecuzione non è lineare come nel codice imperativo classico. - **Troppi oggetti (over-engineering)**: un cattivo utilizzo di DI può portare a frammentare eccessivamente il codice: troppe interfacce, troppe classi per ogni piccola cosa, il che rende il sistema verboso. Bisogna trovare un equilibrio. - **Constructor over-injection**: se troviamo classi con 6-7 dipendenze iniettate nel costruttore, potrebbe essere un odore di design (forse la classe fa troppo). Anche gestire la configurazione di molte dipendenze può diventare complicato.

Inoltre, l'uso di container DI mal configurati può portare a errori runtime difficili da diagnosticare (es. dipendenze non registrate che causano eccezioni). Quindi, DI va applicata con criterio: **troppa injection può essere un antipattern** se non riflette reali necessità architetturali.

```
// Esempio di potenziale anti-pattern: troppi servizi iniettati (constructor
over-injection)
public class OrderManager {
    public OrderManager(IRepository repo, IPaymentService paySvc,
IEmailService emailSvc, ILogger logger, IValidator validator) {
        // ... 5 dipendenze, forse è segno che OrderManager fa troppe
cose ...
    }
    // ...
}
// Una classe con così tante dipendenze può violare SRP. Potrebbe essere
opportuno dividerla in più classi con responsabilità specifiche.
```

Slide 42: DI e Codice Legacy

Introdurre Dependency Injection in codice **legacy** (non pensato per DI) può essere impegnativo ma fattibile con alcuni accorgimenti. Spesso il codice legacy crea oggetti concretamente all'interno di metodi (accoppiamento forte). Per invertire la dipendenza, si possono usare **adapter o wrapper**: si crea un'interfaccia che rappresenta il comportamento dell'oggetto legacy e un adattatore che lo incapsula. In questo modo, il codice nuovo dipenderà dall'interfaccia e potrà ricevere in injection l'adapter che internamente delega al componente legacy.

Esempio: supponiamo di avere una classe legacy statica `LegacyAuth` con un metodo statico `Login`. Possiamo creare un'interfaccia `IAuthService` con un metodo `Login` non statico, poi un adattatore `LegacyAuthAdapter` che implementa `IAuthService` chiamando `LegacyAuth.Login`. Le nuove classi useranno `IAuthService` e potremo iniettare `LegacyAuthAdapter`. Questo permette di testare più facilmente (magari creando anche un fake di IAuthService) e di isolare il codice legacy in un unico punto.

```
// Codice legacy (difficile da testare e da iniettare direttamente):
static class LegacyAuth {
    public static bool Login(string user, string pwd) {
        // ... verifica su un sistema legacy ...
        return (user == "admin" && pwd == "1234");
    }
}

// Nuova interfaccia per astrarre l'autenticazione:
public interface IAuthService {
    bool Login(string username, string password);
}

// Adapter verso il sistema legacy:
public class LegacyAuthAdapter : IAuthService {
    public bool Login(string username, string password) {
        return LegacyAuth.Login(username, password);
    }
}

// Uso nella nuova logica con DI:
public class AccountController {
    private IAuthService _auth;
    public AccountController(IAuthService authService) {
        _auth = authService;
    }
    public void EseguiLogin(string user, string pwd) {
        if(_auth.Login(user, pwd)) Console.WriteLine("Login riuscito");
        else Console.WriteLine("Credenziali errate");
    }
}

// Configurazione: iniettiamo l'adapter legacy come implementazione di
IAuthService
IAuthService auth = new LegacyAuthAdapter();
var controller = new AccountController(auth);
controller.EseguiLogin("admin", "1234");  // utilizza internamente
LegacyAuth.Login tramite l'adapter
```

Slide 43: Strumenti DI in .NET

Nell'ecosistema .NET esistono vari **framework DI/IoC** che facilitano l'adozione di questo pattern: - Il **contenitore nativo di .NET Core** (Microsoft.Extensions.DependencyInjection) è oggi uno standard de-

facto, integrato nei template di ASP.NET Core, supporta lifetimes (Transient/Scoped/Singleton) ed è abbastanza semplice e performante. - **Autofac**: un potente container DI di terze parti, con funzionalità avanzate (module, autoconfiguration, ecc). Spesso usato prima che .NET Core avesse il suo container, rimane popolare per scenari avanzati. - **Ninject**: altro container DI popolare in passato, noto per la sintassi fluida (fluent) nelle configurazioni (es. `kernel.Bind<IFoo>().To<Foo>()`). - **Unity Container**: sviluppato da Microsoft Patterns & Practices, usato molto ai tempi di .NET Framework e applicazioni enterprise. - **Castle Windsor**: un container maturo e ricco di funzionalità, parte del progetto Castle. - Altri: StructureMap, LightInject, DryIoc, etc., ognuno con propri punti di forza (performance, leggerezza, caratteristiche extra).

Questi strumenti semplificano la gestione delle dipendenze e offrono caratteristiche come injection via costruttore, proprietà, metodi, interceptors (per cross-cutting concerns), assembly scanning (registrazione automatica delle interfacce implementate), etc. La scelta spesso dipende dai requisiti e preferenze, ma oggi con .NET Core la tendenza è usare il container integrato e magari estenderlo se serve.

```
/* Esempio di configurazione con Autofac:
var builder = new ContainerBuilder();
builder.RegisterType<ConsoleLogger>().As<ILogger>().SingleInstance();
builder.RegisterType<EmailNotificatore>().As<INotificatore>().InstancePerDependency();
// Autofac consente molte configurazioni fluent, come InstancePerDependency
(Transient), SingleInstance (Singleton), etc.
using(var container = builder.Build()) {
    using(var scope = container.BeginLifetimeScope()) {
        var serv = scope.Resolve<INotificatore>();  // Ottieni l'istanza di
EmailNotificatore come INotificatore
        serv.Invia("Test DI con Autofac");
    }
}
*/
```

Slide 44: Riepilogo Dependency Injection

In conclusione, la **Dependency Injection** è diventata un pilastro della progettazione di applicazioni modulari e scalabili: - **Favorisce l'uso di interfacce** e separa il *cosa fa* un componente dal *come ottiene* ciò che gli serve. Il risultato è un codice più pulito e **manutenibile**, dove le dipendenze sono chiare e facilmente sostituibili. - Si integra perfettamente con i principi SOLID (in particolare DIP e SRP), portando a un'architettura più robusta. DI rende esplicite le responsabilità di configurazione a un livello alto (composizione dell'app), mentre i componenti rimangono focalizzati sui loro compiti. - L'uso combinato di DI e IoC container automatizza gran parte del wiring del sistema, riducendo errori manuali e codice boilerplate. Tuttavia, è importante usare DI quando serve: la semplicità rimane una virtù, quindi evitare di complicare inutilmente disegni semplici.

Applicare correttamente la Dependency Injection richiede un po' di pratica, ma i benefici ripagano: test più facili, codice flessibile, architettura pulita. Riassumendo alcune **buone pratiche**:

```
/*
- Usa il costruttore per le dipendenze obbligatorie (constructor injection).
- Mantieni i costruttori snelli: se sono troppi parametri, valuta di
```

```
refactorizzare.
- Programma contro interfacce e provvedi implementazioni configurabili
esternamente (principio DIP).
- Registra chiaramente le dipendenze in un container o in un modulo di
composizione.
- Evita di mescolare Service Locator e DI se possibile; preferisci DI
esplicita per chiarezza.
- Documenta se qualche dipendenza è opzionale (es. property injection) così
gli utilizzatori sapranno come usarla.
*/
```

Architettura a Microservizi

Slide 45: Architettura a Microservizi – Introduzione

L'**architettura a microservizi** è uno stile architetturale in cui un'applicazione è strutturata come una collezione di **piccoli servizi indipendenti**, ciascuno eseguente in un proprio processo e comunicante con gli altri tramite meccanismi leggeri (spesso API HTTP/REST o code di messaggi). Ogni microservizio è focalizzato su una specifica funzionalità di business (es. gestione ordini, autenticazione, catalogo prodotti) ed è **autonomamente distribuibile**.

A differenza del modello **monolitico** (un'unica applicazione che include tutte le funzionalità), i microservizi isolano i domini: ciò permette di sviluppare, scalare e distribuire ogni servizio in modo indipendente. Ad esempio, se il servizio Ordini richiede più risorse, si può scalare solo quello, senza dover scalare l'intera app. Questa architettura, però, introduce complessità a livello di comunicazione e gestione distribuita. Nei prossimi slide esploreremo comunicazione, orchestrazione e altre sfide dei microservizi.

```
/* Confronto concettuale:
   Monolite: un unico deploy contenente moduli: [Autenticazione + Ordini +
Prodotti + ...] in un processo.
   Microservizi: più deploy separati, es:
      - AuthService (gestione autenticazione) – esposto su http://mioapp.com/
auth
      - OrderService (gestione ordini) – esposto su http://mioapp.com/order
      - ProductService (gestione catalogo prodotti) – esposto su http://
mioapp.com/product
   Questi servizi comunicano tra loro via rete quando necessario.
*/
```

Slide 46: Vantaggi dei Microservizi

I microservizi offrono notevoli **vantaggi** architetturali: - **Scalabilità fine-granulare**: è possibile scalare verticalmente o orizzontalmente solo i servizi critici. Se ad esempio il servizio di ricerca prodotti è molto usato, possiamo eseguire più istanze di *ProductService* senza toccare gli altri. - **Indipendenza di deployment**: ogni microservizio può essere distribuito (deploy) in autonomia. Un team può rilasciare una nuova versione del proprio servizio senza dover riconsegnare l'intera applicazione. Questo abilita cicli di release più rapidi e isolati. - **Resilienza e isolamento dei guasti**: se un microservizio va in errore o risulta non disponibile, l'intera applicazione non necessariamente crolla. Gli altri servizi continuano a

funzionare (magari con funzionalità degradate). Si evitano quindi *Single Point of Failure* monolitici. - **Eterogeneità tecnologica**: ogni servizio può essere sviluppato con la tecnologia/language più adatta al suo problema (purché rispettino protocolli comuni di comunicazione). Ad esempio, un microservizio di Machine Learning potrebbe essere scritto in Python, mentre il resto in C#. - **Manutenibilità e focalizzazione**: codebase più piccole per servizio significano meno complessità locale, facilitando la comprensione. I team possono essere organizzati attorno ai microservizi (autonomi e cross-funzionali), migliorando la produttività.

Questi benefici rendono i microservizi ideali per **grandi sistemi** con esigenze di scalabilità e rapidità di sviluppo.

```
// Esempio: scalare un microservizio specifico (concetto illustrativo con
Docker Compose)
version: '3'
services:
  orderservice:
    image: myapp/orderservice:1.0
    deploy:
      replicas: 3   # Eseguiamo 3 istanze di OrderService per gestire più
carico
  productservice:
    image: myapp/productservice:1.0
    deploy:
      replicas: 1   # ProductService ha meno carico, 1 istanza sufficiente
// Questo mostra che possiamo aumentare repliche di un servizio
indipendentemente dagli altri.
```

Slide 47: Svantaggi dei Microservizi

Accanto ai vantaggi, i microservizi portano anche **sfide e complessità**: - **Complessità architetturale**: un sistema di microservizi è essenzialmente un sistema distribuito. Bisogna gestire la comunicazione tra servizi, la scoperta dei servizi (service discovery), la configurazione distribuita, etc. L'infrastruttura necessaria è più complessa rispetto a un monolite. - **Overhead di comunicazione**: chiamare un altro servizio via rete è più lento di una chiamata in-process. C'è latenza di rete e possibili fallimenti di rete da considerare. Inoltre, la serializzazione/deserializzazione (es. in JSON) aggiunge overhead. - **Consistenza dei dati**: in un monolite spesso c'è un singolo database. Con microservizi, ogni servizio tende ad avere il proprio database, quindi mantenere la consistenza tra dati distribuiti è difficile. Di solito si adotta la **consistenza eventuale** e meccanismi come eventi per sincronizzare. - **Debug e testing**: tracciare un flusso che passa per più servizi è complicato. Serve implementare **tracciamento distribuito** e centralizzare i log. Anche i test integrati diventano più complessi (bisogna far girare molti servizi in concerto). - **DevOps e distribuzione**: passare a microservizi richiede un forte investimento in automazione di deployment (CI/CD), containerizzazione (Docker), orchestrazione (Kubernetes, Docker Swarm) e monitoraggio. L'operatività è più onerosa: c'è bisogno di gestire decine di unità invece di una. - **Duplica di codice e interfacce**: occorre definire chiaramente le API tra servizi. Cambiare un contratto di servizio impatta i client, richiedendo coordinamento. Inoltre funzionalità trasversali (es. autenticazione) potrebbero dover essere implementate in ogni servizio o centralizzate tramite gateway.

In breve, i microservizi **non sono una pallottola d'argento**: la loro adozione va giustificata dal contesto (solitamente progetti grandi o necessità di scalabilità massiva).

```
// Esempio di overhead: chiamata di rete vs chiamata locale
// Chiamata locale (monolite):
var prezzo = CalcolaPrezzo(carrello);  // chiamata metodo in-process,
nanosecondi

// Chiamata microservizio via rete:
HttpClient client = new HttpClient();
string risposta = await client.GetStringAsync("http://priceservice/api/
calcola?carrelloId=123");
// Questa chiamata può impiegare millisecondi o più, introdurre latenza e
possibili errori di rete.
// Monitored output: latenza network ~50ms, plus elaborazione server.
```

Slide 48: Comunicazione tra Microservizi

Nei microservizi, la **comunicazione** tra servizi è cruciale. Due modelli principali: - **Sincorno (tipicamente HTTP/REST)**: un servizio chiama direttamente l'API di un altro, aspettando la risposta. Ad esempio OrderService fa una richiesta REST a ProductService per ottenere informazioni di un prodotto. Questo modello è semplice ma introduce accoppiamento temporale (il chiamante deve attendere il chiamato) e richiede che i servizi siano disponibili simultaneamente. - **Asincrono (messaggistica/eventi)**: i servizi comunicano tramite code di messaggi o broker di eventi (es. RabbitMQ, Kafka, Azure Service Bus). Un servizio pubblica un evento (es. "Ordine creato") e uno o più altri servizi lo consumano reagendo di conseguenza. Oppure un servizio invia un messaggio a una coda per essere elaborato da un altro servizio in background. Questo decoupling temporale aumenta la resilienza (il messaggio può essere elaborato quando il consumer è pronto) ma aggiunge complessità (gestione di code, processi idempotenti, ordine dei messaggi).

Spesso si usa una combinazione: ad esempio, query sincronhe per ottenere dati immediati, ma pattern asincroni (pub/sub) per propagare aggiornamenti di stato o orchestrare processi che possono essere eventuali. La scelta dipende dalle esigenze di consistenza e latenza. In ogni caso, la definizione chiara delle API (contratti) è fondamentale: di solito REST con JSON per sincro, e formati come JSON/Avro/Protobuf per messaggi.

```
// Esempio di chiamata sincrona REST da un microservizio a un altro in C#:
using var client = new HttpClient();
var response = await client.GetAsync("http://productservice/api/prodotti/
42");
if(response.IsSuccessStatusCode) {
    string json = await response.Content.ReadAsStringAsync();
    Console.WriteLine("Dettagli prodotto JSON: " + json);
}

// Esempio di invio asincrono di messaggio (pseudo-codice):
var orderCreatedEvent = new OrderCreatedEvent { OrderId = 123, CustomerId =
456 };
messageBus.Publish(orderCreatedEvent);  // pubblica un evento che sarà
gestito da altri servizi (es. EmailService, ShippingService)
```

Slide 49: Orchestrazione vs Coreografia

Nel contesto di microservizi, due approcci per gestire flussi di lavoro distribuiti sono: - **Orchestrazione**: un servizio centralizzato (orchestrator) coordina le chiamate ai vari microservizi per portare a termine un processo. L'orchestrator conosce il flusso completo (ad esempio, per processare un ordine: chiama PaymentService, poi InventoryService, poi ShippingService in sequenza). I vantaggi: la logica del processo è in un solo posto e si può gestire facilmente lo stato e compensazioni. Svantaggi: l'orchestrator diventa un componente critico e aumenta l'accoppiamento (sa di tutti gli altri servizi). - **Coreografia**: non c'è un coordinatore centrale; i microservizi **si parlano attraverso eventi** reagendo gli uni agli altri. Ad esempio, OrderService emette "Order Created", PaymentService ascolta e se il pagamento va a buon fine emette "Payment OK", InventoryService ascolta e riserva stock, etc. Qui ogni servizio conosce solo gli eventi di cui ha bisogno. Vantaggio: massima decoupling, ogni servizio è autonomo. Svantaggio: la logica di business complessiva è dispersa tra servizi ed eventi, diventando difficile da tracciare (emergent design) e potenzialmente con cicli di feedback complessi.

Entrambi gli approcci possono coesistere: per processi semplici, la coreografia via eventi può bastare; per processi transazionali complessi, un orchestratore (ad esempio implementando il pattern Saga) può semplificare gestione di rollback. L'importante è assicurarsi che i servizi rimangano il più possibile indipendenti nei loro compiti.

```
/* Esempio di flusso Orchestrazione vs Coreografia:

Orchestrazione (centrale):
 Orchestrator -> PaymentService.ProcessaPagamento(ordine)
 Orchestrator -> InventoryService.RiservaStock(ordine)
 Orchestrator -> ShippingService.ProgrammaSpedizione(ordine)
 (L'orchestrator attende le risposte e decide i passi successivi)

Coreografia (eventi):
 OrderService emette evento OrdineCreato(ordineId)
 PaymentService ascolta OrdineCreato, esegue pagamento, emette evento
PagamentoApprovato(ordineId)
 InventoryService ascolta PagamentoApprovato, riserva stock, emette evento
StockRiservato(ordineId)
 ShippingService ascolta StockRiservato, programma spedizione (fine processo)
 (Ogni servizio reagisce agli eventi senza un coordinatore unico)
*/
```

Slide 50: Pattern Saga (Orchestrazione Distribuita)

Il **pattern Saga** è una soluzione per gestire transazioni distribuite nei microservizi, mantenendo consistenza eventuale tramite una serie di passi locali ciascuno con una possibile azione di **compensazione** in caso di fallimento di uno step successivo. In pratica, una Saga è una sequenza di operazioni in vari servizi: se tutti riescono, ok; se uno fallisce, si eseguono operazioni inverse (compensazioni) per annullare gli effetti dei passi già completati.

Le Saga possono essere implementate in due modi: - **Orchestrated Saga**: un orchestrator gestisce lo stato della saga e dice ai servizi quando compiere l'azione e quando compiere eventuale rollback. (Segue architettura orchestrazione). - **Choreography Saga**: non c'è orchestratore; i servizi coinvolti

emettono eventi di successo o fallimento, e altri servizi reagiscono eseguendo compensazioni se necessario.

Ad esempio, Saga per "Creazione ordine": OrderService crea un ordine nello stato "Pending". Orchestrator chiede a PaymentService di addebitare; se pagamento ok, chiede a InventoryService di riservare stock; se uno dei due fallisce, chiede al precedente di compensare (storno pagamento se stock fallisce, o rilascia stock se spedizione fallisce, ecc.) e marca l'ordine come "Failed". Questo garantisce che non rimangano operazioni parziali incoerenti.

```
// Pseudo-codice di orchestrazione Saga (sincrona per semplicità):
async Task<string> ProcessOrderSaga(int orderId) {
    bool paymentOk = await PaymentService.Paga(orderId);
    if (!paymentOk) {
        return "Ordine ANNULLATO: pagamento fallito";
    }
    bool stockOk = await InventoryService.RiservaStock(orderId);
    if (!stockOk) {
        await PaymentService.Rimborso(orderId);  // Compensazione: annulla
pagamento
        return "Ordine ANNULLATO: stock non disponibile, pagamento
rimborsato";
    }
    bool shipmentOk = await ShippingService.Spedisci(orderId);
    if (!shipmentOk) {
        await InventoryService.LiberaStock(orderId);  // Compensazione:
libera stock
        await PaymentService.Rimborso(orderId);       // Compensazione:
annulla pagamento
        return "Ordine ANNULLATO: spedizione fallita, ripristinati pagamento
e stock";
    }
    return "Ordine COMPLETATO con successo";
}
```

Slide 51: API Gateway

In una architettura a microservizi, è comune introdurre un **API Gateway** come punto di ingresso unico per le richieste dei client. L'API Gateway è un servizio che **orchestra e instrada** le chiamate verso i microservizi interni appropriati. I client (es. applicazioni front-end, mobile) si interfacciano solo con il gateway, che espone un set unificato di endpoint, dietro ai quali nasconde la composizione dei servizi sottostanti.

Vantaggi dell'API Gateway: - **Nasconde la complessità interna**: il client non deve conoscere tutti gli URL dei microservizi né orchestrare chiamate multiple; il gateway può anche aggregare risposte da più servizi in una singola risposta per il client. - **Sicurezza e cross-cutting**: autenticazione, autorizzazione, rate limiting, caching, logging centralizzato possono essere gestiti nel gateway, alleggerendo i singoli servizi. - **Adattamento protocolli**: può esporre un protocollo differente ai client (es. GraphQL, gRPC-web) e tradurre in REST interni, o viceversa.

Lo svantaggio è che il gateway diventa un componente critico e deve essere altamente disponibile e scalabile. Tecnologie comuni: Ocelot (gateway per .NET), YARP (reverse proxy per .NET), Kong, API Management (Azure), AWS API Gateway, etc.

```
// Esempio concettuale: definizione di routing in un API Gateway (pseudo-
csharp)
app.Map("/api/ordini/{**rest}", async context => {
    // Instrada tutte le richieste /api/ordini/* al servizio Ordini interno
    var targetUrl = "http://orderservice/" +
context.Request.Path.Value.Substring("/api/ordini/".Length);
    var response = await httpProxy.Forward(context, targetUrl);
    // Il gateway potrebbe aggiungere header (es. auth) o trasformare la
risposta se necessario.
});

app.Map("/api/clienti/{id}", async context => {
    // Esempio di aggregazione: richiesta cliente con ultimi ordini
    string id = context.GetRouteValue("id");
    var clienteJson = await httpProxy.GetStringAsync($"http://
customerservice/api/clienti/{id}");
    var ordiniJson = await httpProxy.GetStringAsync($"http://orderservice/
api/ordini?clienteId={id}");
    // Combina i dati di cliente e ordini in un'unica risposta JSON
    context.Response.ContentType = "application/json";
    await context.Response.WriteAsync($"{{ \"cliente\": {clienteJson},
\"ordini\": {ordiniJson} }}");
});
```

Slide 52: Archiviazione dei Dati nei Microservizi

Nei microservizi si adotta spesso il principio **"Database per microservizio"**, ovvero ogni servizio gestisce un proprio archivio dati, evitando di condividerlo con altri. Questo riduce accoppiamento a livello di schema e consente a ciascun servizio di usare il tipo di database più adatto (relazionale, NoSQL, ecc.). Ad esempio, ProductService può avere un database SQL per il catalogo prodotti, mentre OrderService usa un database NoSQL per gli ordini.

Il rovescio della medaglia è che dati duplicati o replicati possono esistere (denormalizzazione tra servizi) e bisogna garantire consistenza con meccanismi applicativi (eventi, sincronizzazioni). L'assenza di transazioni distribuite forti spinge verso modelli eventual consistent. Pattern come **CQRS** (separare comando e query) e **Event Sourcing** talvolta sono adottati per mantenere storici di eventi con cui ricostruire lo stato in diversi servizi.

Quando serve che un servizio legga dati di un altro, spesso lo fa chiamando l'API o mantenendo una copia locale aggiornata via eventi. È importante evitare l'errore di far accedere più servizi allo stesso schema di database, altrimenti si perde l'isolamento.

```
/* Esempio di separazione dei database:
   - UserService -> Database Utenti (SQL Server)
   - OrderService -> Database Ordini (MongoDB)
```

```
   - InventoryService -> Database Magazzino (PostgreSQL)

   Scenario: Quando un ordine è creato, OrderService scrive su MongoDB e
emette un evento "OrderCreated".
   InventoryService ascolta l'evento, riserva il magazzino e scrive sul
proprio DB Postgres lo stato.
   Se OrderService ha bisogno di info utente, chiama UserService invece di
leggere direttamente il DB utenti.
   Ogni servizio è padrone dei propri dati.
*/
```

Slide 53: Contenitori e Deploy dei Microservizi

La proliferazione di microservizi richiede strumenti per **distribuire e gestire** decine o centinaia di servizi. Qui entrano in gioco i **container** (es. Docker) e gli orchestratori come **Kubernetes**: - **Docker/ Container**: impacchetta un microservizio con il suo ambiente (runtime, librerie) in un'unità isolata. Questo garantisce consistenza: il servizio gira allo stesso modo in sviluppo, staging e produzione. Facilita anche il scaling, creando più istanze container identiche. - **Kubernetes (K8s)**: sistema per orchestrare container su cluster di macchine. Gestisce il deployment dichiarativo (ad es. "voglio 5 istanze di OrderService"), auto-riparazione (riavvia container crashati), bilanciamento di carico tra istanze, aggiornamenti rolling (update graduale delle versioni), e molto altro. In microservizi è praticamente lo standard per gestire l'infrastruttura.

Ciò significa che oltre a scrivere codice, il team deve abbracciare pratiche DevOps: pipeline CI/CD per costruire immagini container e file di configurazione per orchestratori. L'uso di container rende anche possibile sfruttare servizi cloud (AWS ECS/EKS, Azure AKS) per semplificare il deployment.

```
// Esempio di Dockerfile per un microservizio .NET:
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS runtime
WORKDIR /app
COPY ./publish_output/ ./
# Imposta variabili di configurazione se necessario
ENV ASPNETCORE_URLS=http://+:80
ENTRYPOINT ["dotnet", "OrderService.dll"]

/* Questo Dockerfile prende l'app pubblicata di OrderService e la confeziona
in un container.
   Su Kubernetes, potremmo avere un file YAML di Deployment per eseguire X
repliche di quest'immagine,
   e un Service per esporle sulla rete cluster (o tramite ingress).
*/
```

Slide 54: Osservabilità nei Microservizi

Con tanti servizi in esecuzione, diventa fondamentale implementare **osservabilità**: - **Logging centralizzato**: invece di controllare log separatamente su ogni servizio/istanza, si utilizzano sistemi che aggregano i log (es. ELK stack: Elasticsearch + Logstash + Kibana, o servizi cloud). Ogni log include magari un identificatore di **correlazione** (trace ID) per collegare le richieste tra servizi. - **Distributed Tracing**: strumenti come OpenTelemetry, Jaeger, Zipkin permettono di tracciare una singola transazione

mentre attraversa diversi microservizi. Iniettano un ID di traccia in ogni richiesta e misurano tempi e eventuali errori in ciascun segmento, producendo una timeline visuale utile per debug e performance tuning. - **Metrics e Monitoring**: estrarre metriche (es. tempo di risposta, throughput, utilizzo CPU/memoria) e allarmi. Tools come Prometheus (per collezionare metriche da servizi) e Grafana (per dashboard) consentono di monitorare lo stato di un sistema microservizi in tempo reale. Ad esempio, quanti ordini processati al minuto, quanti errori 500 su PaymentService, etc. - **Health check e alerting**: i microservizi spesso espongono endpoint di health (`/healthz`) che indicano se il servizio è funzionante. L'orchestratore o il sistema di monitoring li interroga per prendere decisioni (riavviare container non sani, notificare team on-call).

Implementare osservabilità robusta richiede aggiungere codice nei servizi (per log/tracing) e configurare l'infrastruttura di supporto, ma è indispensabile per gestire la complessità.

```
// Esempio: utilizzo di Correlation ID per logging
string correlationId = Guid.NewGuid().ToString();
_logger.Log($"[Req {correlationId}] Ordine ricevuto, inizio elaborazione");
// ... chiamata a PaymentService con header "X-Correlation-ID:
{correlationId}" ...
_logger.Log($"[Req {correlationId}] Ordine completato correttamente");
// In questo modo, tutti i log relativi a una certa richiesta condividono un
ID,
// e mediante uno strumento di aggregazione possiamo filtrare tutti i log di
quella richiesta attraverso i servizi.
```

Slide 55: Conclusioni Microservizi

L'architettura a microservizi rappresenta un **approccio potente** per costruire applicazioni enterprise flessibili e scalabili, ma comporta un aumento di complessità infrastrutturale e progettuale. È importante valutare caso per caso: microservizi sono ideali quando si prevede di dover scalare componenti in modo indipendente, avere team dedicati a diversi moduli e rilasci frequenti. In contesti più piccoli o con dominio non ben suddivisibile, una soluzione monolitica modulare può essere più semplice e adeguata.

In sintesi: - **Pro**: scalabilità indipendente, resilienza ai guasti, rapidità di sviluppo parallelo, allineamento con organizzazione dei team, adozione di tecnologie diversificate per sottoproblemi. - **Contro**: richiede investimenti in **DevOps, monitoring e orchestrazione**. Aumenta la difficoltà di debug e di mantenere la coerenza. Può portare a duplicazioni e aumentare il consumo di risorse (più istanze, overhead di rete). - **Best practice**: definire chiari **bounded context** per i servizi (ogni microservizio con un perimetro di responsabilità ben definito), automatizzare tutto (CI/CD), implementare robusta osservabilità e fallback (circuit breakers, retry, ecc.), e adottare standard di comunicazione ben definiti.

I microservizi non sono una destinazione obbligata, ma uno strumento architetturale. Usarli al momento giusto può dare grandi benefici; usarli a sproposito può complicare inutilmente un progetto. La chiave è sempre bilanciare **flessibilità** e **complessità**.

```
/* Riepilogo Microservizi:
   - Strutturare per dominio: ogni servizio fa una cosa (Single
Responsibility a livello architettura).
   - Evitare eccessiva interdipendenza: preferire comunicazione tramite
```

```
eventi dove possibile per decoupling.
   - Pianificare DevOps: container, orchestrator, pipeline di test e
deployment.
   - Monitorare e adattare: una volta in produzione, analizzare metriche e
log per migliorare interazioni e performance.
   - Ricordare che la migrazione a microservizi è graduale: monolite modulare
-> estrazione graduale di servizi.
*/
```
