Slide 1 – Test-Driven Development in C# (Intermedio)
- Introduzione al **Test-Driven Development (TDD)** applicato a C#
- Lezione di livello **intermedio** (richiesta familiarità con la sintassi di base di C#)
- Obiettivo: imparare a progettare e sviluppare codice in C# guidato dai test automatici (unit test)

Slide 2 – Agenda della Lezione
- **Panoramica del TDD:** cos'è il Test-Driven Development e quali vantaggi offre
- **Framework xUnit:** come scrivere e eseguire test unitari in C# con xUnit
- **Esempi pratici TDD:** sviluppo guidato dai test su casi generici con codice C#
- **Design "test-oriented":** principi di progettazione (SRP, DI) per codice testabile
- **Refactoring e Mocking:** migliorare il codice in sicurezza e isolare dipendenze con *mock* (es. libreria Moq)
- **Best practice:** consigli di stile, organizzazione del codice di produzione e di test

Slide 3 – Cos'è il Test-Driven Development (TDD)
- **Sviluppo guidato dai test:** metodologia in cui **si scrivono prima i test** automatici e poi il codice di produzione [1]
- Il ciclo tipico è: **scrivi un test che fallisce**, implementa il minimo codice per farlo passare, quindi **refactoring** del codice [2] [3]
- TDD integra sviluppo e testing: test e codice vengono scritti insieme in brevi iterazioni, riducendo i tempi di debug e migliorando la qualità complessiva

Slide 4 – Ciclo Red-Green-Refactor
Il ciclo TDD in tre fasi: Rosso – scrivi un test e verificane il fallimento; Verde – scrivi codice minimo per far passare il test; Refactor – migliora il design del codice mantenendo i test verdi. Questo ciclo si ripete per ogni nuova funzionalità. [4] [5]

Slide 5 – Fasi del Ciclo TDD
- **Red (Rosso):** scrivi un nuovo test automatico per una funzionalità *non ancora implementata*. Esegui tutti i test e verifica che **il nuovo test fallisca** (indicando che serve nuovo codice) [6] [3]
- **Green (Verde):** implementa rapidamente **solo il codice necessario** a far passare il test. Non cercare soluzioni perfette in questa fase, punta a soddisfare le asserzioni del test
- **Refactor:** con tutti i test verdi, **riorganizza e pulisci il codice** di produzione *e* dei test. Migliora nomi, elimina duplicazioni, applica design pulito mantenendo i test tutti positivi [7]

Slide 6 – Vantaggi del TDD (I)
- **Meno bug:** scrivendo test prima del codice, si individuano edge case e requisiti prima di implementare. Ciò previene molti difetti e regressioni fin dalle prime fasi [8] [9]
- **Qualità del codice migliorata:** il TDD incoraggia a progettare codice a piccole unità focalizzate (funzioni e classi semplici). Ogni unità viene verificata in isolamento, favorendo un design più pulito e modulare [10] [11]
- **Feedback immediato:** la suite di test automatizzati fornisce un riscontro rapido dopo ogni modifica. Ogni errore introdotto viene subito rilevato, rendendo il processo di sviluppo più efficiente

Slide 7 – Vantaggi del TDD (II)
- **Manutenibilità e confidenza:** una base di test completa funge da **rete di sicurezza** durante l'evoluzione del codice. Si può refattorizzare o aggiungere funzionalità con la sicurezza che i test catturino eventuali malfunzionamenti [12] [13]

- **Documentazione vivente:** i test descrivono in modo eseguibile il comportamento atteso del sistema. Un nuovo sviluppatore può leggere i test per capire come dovrebbero funzionare le classi/metodi
- **Design orientato all'utilizzo:** scrivere prima i test spinge a pensare dal punto di vista dell'**utilizzatore dell'API** (public interface). Questo porta a interfacce più intuitive e a classi con singole responsabilità [14]

Slide 8 – Introduzione a xUnit
- **xUnit.net** è un framework open source per unit testing in .NET, parte della famiglia xUnit (evoluzione di NUnit)
- Fornisce attributi come `[Fact]` e `[Theory]` per definire test, e un ricco insieme di **asserzioni** (`Assert`) per verificare i risultati
- Integrazione facile con .NET Core/5+: si può eseguire i test via riga di comando (`dotnet test`) o tramite test runner di Visual Studio. xUnit supporta l'esecuzione **parallela** e l'isolamento automatico dei test (istanza separata per ogni test) [15]

Slide 9 – Creare un Progetto di Test con xUnit
- In Visual Studio: Aggiungere un nuovo progetto di tipo "xUnit Test Project". Questo include i pacchetti NuGet necessari (`xunit`, `xunit.runner.visualstudio`, `Microsoft.NET.Test.Sdk`)
- Da CLI .NET: eseguire `dotnet new xunit -o NomeProgetto.Tests` per creare un progetto di test xUnit [16]. Aggiungere poi un riferimento al progetto con il codice da testare (`dotnet add NomeTest.csproj reference ../NomeProd.csproj`) [17]
- Organizzazione: per convenzione i test risiedono in un progetto separato (es. *MyApp.Tests*), con struttura di namespace parallela al progetto principale. In questo modo i test possono avere accesso alle classi pubbliche da verificare

Slide 10 – Primo Test con xUnit ([Fact])
- Un test in xUnit è un semplice metodo C# contrassegnato con l'attributo `[Fact]`. Esempio di un test elementare:

```
public class CalcolatriceTests {
    [Fact]  // indica che è un test
    public void Somma_2e3_Restituisce5() {
        var calc = new Calcolatrice();
        int risultato = calc.Somma(2, 3);
        Assert.Equal(5, risultato);
    }
}
```

  - **Arrange-Act-Assert:** nel test sopra, si prepara l'oggetto (`Arrange`), si esegue l'operazione da testare (`Act`) e si verifica il risultato atteso (`Assert`)
  - Se eseguiamo questo test, xUnit avvierà il metodo `Somma_2e3_Restituisce5`. Se l'asserzione (`5 == risultato`) è vera, il test passa; altrimenti fallisce indicando il valore atteso vs. attuale

Slide 11 – Asserzioni Comuni in xUnit
- **Assert.Equal(expected, actual):** verifica che `actual` sia uguale a `expected` (per tipi value o oggetti se opportunamente *equi*)
- **Assert.True(condizione) / Assert.False(condizione):** verificano che una condizione booleana sia rispettivamente vera o falsa
- **Assert.Null(obj) / Assert.NotNull(obj):** verifica che un riferimento sia (o non sia) nullo
- **Assert.Contains(item, collection) / Assert.Throws<ExceptionType>(action):** verifica che una

collezione contenga un elemento, oppure che l'azione passata lanci un'eccezione specifica
- xUnit offre molte altre asserzioni (ad es. `Assert.StartsWith` per stringhe, `Assert.InRange` per valori numerici). L'uso appropriato di queste aiuta a rendere i test chiari e specifici

Slide 12 – Testare le Eccezioni con xUnit
- Per verificare che un metodo lanci una certa **eccezione**, si utilizza `Assert.Throws<T>` passando una lambda che esegue il metodo sotto test
- Esempio: testare che una divisione per zero lanci la giusta eccezione:

```
[Fact]
public void Dividi_DivisionePerZero_LanciaEccezione() {
    var calc = new Calcolatrice();
    // Verifica che Dividi(10,0) lanci DivideByZeroException
    Assert.Throws<DivideByZeroException>(() => calc.Dividi(10, 0));
}
```

- In questo caso il test passa se la chiamata `calc.Dividi(10,0)` genera una `DivideByZeroException`. Se non viene lanciata alcuna eccezione (o ne viene lanciata un'altra), il test fallisce
- Le asserzioni su eccezioni aiutano a testare i **casi di errore** e comportamenti anomali, assicurando che il codice gestisca correttamente condizioni eccezionali

Slide 13 – Test Parametrizzati con [Theory]
- Oltre ai `[Fact]` (test "costanti"), xUnit supporta test **parametrizzati** detti `[Theory]`. Un `[Theory]` permette di eseguire lo stesso test con diversi input e aspettative
- Si usano insieme gli attributi `[Theory]` e `[InlineData(...)]` per specificare una serie di dati di ingresso. Esempio: test per verificare se un numero è dispari:

```
[Theory]
[InlineData(3, true)]
[InlineData(4, false)]
[InlineData(7, true)]
public void NumeroDispari_RitornaAtteso(int numero, bool atteso) {
    bool risultato = Utils.ÈDispari(numero);
    Assert.Equal(atteso, risultato);
}
```

- In questo caso il framework esegue il metodo di test tre volte, passando di volta in volta i valori indicati in `[InlineData]`. Se **tutte** le iterazioni passano (tutte le asserzioni sono vere), il test complessivo è considerato positivo [18] [19]
- I Theory aiutano a evitare duplicazione di test simili e a coprire rapidamente più casi con poco codice

Slide 14 – Esecuzione dei Test (runner)
- I test xUnit possono essere eseguiti con il comando CLI `dotnet test`, che compila il progetto e lancia tutti i test, riportando per ognuno esito e durata [20]
- In Visual Studio, si usa **Test Explorer**: una finestra dedicata elenca tutti i test scoperti. È possibile eseguire tutti o alcuni test, vedere quali passano (verde) o falliscono (rosso) e leggere messaggi di errore/stack trace

- Ogni volta che il codice viene modificato, è buona pratica **rieseguire rapidamente i test**. VS offre anche “Live Unit Testing” (nelle edizioni Enterprise) che esegue continuamente i test in background ad ogni salvataggio
- Integrare i test nel flusso di sviluppo permette di rilevare immediatamente regressioni. Un test fallito indica che qualcosa nel codice non soddisfa più i requisiti attesi, fornendo un *feedback* prezioso durante la scrittura del codice

Slide 15 – Esempio TDD: Implementare uno Stack
- Scenario generico: vogliamo sviluppare una classe `Stack<T>` semplificata (struttura LIFO - Last In First Out) usando TDD, senza fare assunzioni sul dominio applicativo
- **Requisiti:** lo stack deve permettere di inserire elementi (`Push`) e rimuoverli restituendo l’ultimo inserito (`Pop`). Inoltre, `Pop` su uno stack vuoto deve provocare un’eccezione
- Iniziamo applicando il ciclo TDD: scriveremo un test alla volta e il codice minimo per superarlo, iterando finché soddisfiamo tutti i requisiti
- Si assume di avere già una classe vuota `Stack<T>` con metodi da implementare. Procederemo aggiungendo test e codice gradualmente (Red-Green-Refactor) per completare le funzionalità dello stack

Slide 16 – Test 1: Pop su Stack Vuoto

```
[Fact]
public void Pop_SuStackVuoto_LanciaInvalidOperation() {
    var stack = new Stack<int>();
    // Act & Assert: il Pop su uno stack vuoto deve lanciare un’eccezione
    Assert.Throws<InvalidOperationException>(() => stack.Pop());
}
```

- Abbiamo scritto il primo test che descrive il comportamento atteso: se chiamiamo `Pop()` su uno stack vuoto, dovremmo ottenere un’eccezione (usiamo `InvalidOperationException` in questo caso)
- Eseguiamo i test: questo nuovo test naturalmente **fallisce** perché `Stack<T>.Pop()` non è ancora implementato (o è vuoto). Siamo nello stato **Rosso** del ciclo TDD [3]
- Prossimo passo: implementare il minimo codice in `Pop()` per far passare questo test (stato Verde), mantenendo eventualmente fallire tutti gli altri test (che al momento non esistono)

Slide 17 – Implementazione Minima per Test 1
- Per far passare il Test 1, implementiamo `Stack<T>.Pop()` in modo che lanci sempre un’eccezione `InvalidOperationException`. Questo soddisfa il caso di stack vuoto, anche se è una soluzione provvisoria

```
public class Stack<T> {
    // … (altre parti della classe)
    public T Pop() {
        throw new InvalidOperationException("Stack vuoto");
```

```
    }
}
```

- Ora rieseguiamo i test: il Test 1 dovrebbe **passare** (lo stack vuoto lancia effettivamente l'eccezione attesa). Tutti i test sono verdi, possiamo procedere
- Notare che abbiamo scritto **il codice più semplice possibile** per soddisfare il test: al momento `Pop()` non gestisce alcun elemento, ma va bene così finché non scriveremo test aggiuntivi (YAGNI – You Aren't Gonna Need It)

Slide 18 – Test 2: Push e Pop di un Elemento

```
[Fact]
public void PushEPop_UnSoloElemento_RestituisceQuelElemento() {
    var stack = new Stack<string>();
    stack.Push("ciao");
    string risultato = stack.Pop();
    Assert.Equal("ciao", risultato);
}
```

- Secondo scenario: inseriamo un elemento `"ciao"` nello stack e poi facciamo `Pop()`. Ci aspettiamo di riottenere proprio quell'elemento. Inoltre, dopo il Pop lo stack dovrebbe tornare vuoto
- Eseguiamo i test: **Test 2 fallisce**, perché la nostra attuale implementazione di `Pop()` lancia sempre eccezione e non conosce l'elemento inserito. Siamo di nuovo nello stato Rosso
- Ora scriviamo il codice minimo in `Stack<T>` per far passare anche questo test, mantenendo il precedente in verde. Probabilmente dovremo introdurre una struttura dati interna per memorizzare l'elemento inserito con `Push`

Slide 19 – Implementazione per far passare Test 2
- Modifichiamo la classe `Stack<T>` per gestire almeno un elemento:

```
public class Stack<T> {
    private T _soloElemento;
    private bool _haElemento = false;
    public void Push(T item) {
        _soloElemento = item;
        _haElemento = true;
    }
    public T Pop() {
        if (!_haElemento)
            throw new InvalidOperationException("Stack vuoto");
        _haElemento = false;
        return _soloElemento;
    }
}
```

- **Spiegazione:** abbiamo usato due variabili: `_soloElemento` per salvare l'elemento inserito e `_haElemento` (flag) per tracciare se lo stack contiene un elemento. `Push` salva il valore e

marca lo stack come non vuoto, `Pop` controlla il flag, lancia eccezione se vuoto, altrimenti restituisce `_soloElemento` e resetta lo stato a vuoto
- Rilanciamo i test: Test 1 (stack vuoto) rimane positivo (in caso di `_haElemento` false, continua a lanciare eccezione) e **Test 2 ora passa** (inserendo un elemento, Pop restituisce correttamente `"ciao"`). Abbiamo di nuovo tutti i test verdi

Slide 20 – Test 3: Ordine LIFO con più Elementi

```
[Fact]
public void PushMultipli_PopRitornaUltimoInserito() {
    var stack = new Stack<int>();
    stack.Push(5);
    stack.Push(7);
    int primoPop = stack.Pop();
    int secondoPop = stack.Pop();
    Assert.Equal(7, primoPop);
    Assert.Equal(5, secondoPop);
}
```

- Terzo scenario: verifichiamo la proprietà LIFO (Last In, First Out). Inseriamo due valori (5 poi 7) e facciamo due Pop: il primo Pop deve dare 7 (ultimo inserito), il secondo Pop deve dare 5 (il primo inserito, rimasto dopo il primo Pop)
- Con l'implementazione attuale, questo **test fallirà**: il secondo `Push(7)` sovrascrive `_soloElemento`. Al primo `Pop()` otteniamo 7, ma al secondo `Pop()` lo stack risulta vuoto e lancia eccezione invece di restituire 5
- Siamo di nuovo in fase **Rosso**. Per far passare anche questo test dovremo migliorare la struttura interna dello stack, probabilmente usando una collezione (es. una lista) per gestire un numero arbitrario di elementi

Slide 21 – Implementazione per far passare Test 3
- Refactoring della classe `Stack<T>` per gestire una **collezione** di elementi e rispettare l'ordine LIFO:

```
public class Stack<T> {
    private List<T> _elementi = new List<T>();
    public void Push(T item) {
        _elementi.Add(item);
    }
    public T Pop() {
        if (_elementi.Count == 0)
            throw new InvalidOperationException("Stack vuoto");
        int lastIndex = _elementi.Count - 1;
        T item = _elementi[lastIndex];
        _elementi.RemoveAt(lastIndex);
        return item;
```

```
    }
}
```

- Ora `_elementi` contiene tutti gli elementi inseriti. `Push` aggiunge in fondo alla lista. `Pop` controlla se la lista è vuota (come prima), altrimenti preleva l'ultimo elemento inserito (indice `Count-1`), lo rimuove dalla lista e lo restituisce
- Con questa implementazione più robusta, rieseguiamo i test: Test 1, 2 e 3 ora devono **passare tutti**. Abbiamo soddisfatto i requisiti funzionali dello stack mantenendo la suite di test verde. Siamo pronti per un ultimo refactoring e per considerare la funzionalità completa

Slide 22 – Refactoring Finale dello Stack
- Dopo aver soddisfatto i test, possiamo fare un piccolo **refactoring** conclusivo per pulire il codice (ad esempio, il codice della Slide 21 è già piuttosto pulito, ma potremmo ottimizzare nomi o estrarre metodi se necessario)
- Verifichiamo che non ci sia codice duplicato o inutile: in questo caso la rimozione dell'ultimo elemento serve sia per leggere che per aggiornare lo stato, quindi va bene così
- I test sono tutti verdi, garantendoci che le modifiche non abbiano rotto nulla. Abbiamo ottenuto una classe `Stack<T>` funzionante e ben progettata, con copertura di test per casi base e limite
- **Conclusione dell'esempio:** questo processo ha mostrato come TDD porti ad implementare gradualmente la soluzione corretta. Abbiamo iniziato con soluzioni minimali (anche naive) e grazie ai test aggiuntivi siamo arrivati a una soluzione generale, effettuando refactoring in sicurezza man mano che i test assicuravano la correttezza

Slide 23 – Best Practice TDD
- **Procedere per piccoli passi:** scrivere un test alla volta, relativo a una singola funzionalità o caso. Evitare di implementare troppo in una sola iterazione. Ogni ciclo Red-Green-Refactor dovrebbe durare pochi minuti
- **Un solo obiettivo per test:** ogni test dovrebbe verificare un aspetto specifico del comportamento. Test piccoli e mirati sono più leggibili e isolano meglio le cause di fallimento [21] [22]
- **Prima il caso semplice poi i complessi:** inizia dai casi base ("happy path") e poi aggiungi test per edge case ed errori. Questo ti consente di stabilire prima il comportamento fondamentale e poi gestire scenari atipici [23] [24]
- **Mantieni i test veloci e automatici:** una suite di test rapida (<1-2 minuti) incoraggia ad eseguirla spesso. Evita nei test accessi lenti (DB, rete) se non strettamente necessario. Un feedback immediato mantiene il ciclo TDD efficiente [25]

Slide 24 – Principio di Singola Responsabilità (SRP)
- Il **Single Responsibility Principle** afferma che ogni classe o modulo dovrebbe avere **una ed una sola responsabilità**. In altre parole, una classe dovrebbe esistere per un unico scopo o funzionalità
- **Perché aiuta nei test?** Una classe con singola responsabilità tende ad avere meno interdipendenze e meno combinazioni di stati, quindi è **più facile da coprire con test unitari** mirati [26]. I test possono focalizzarsi su un comportamento alla volta
- SRP spesso implica classi più piccole e coese. Questo riduce anche il rischio che una modifica in una parte della classe rompa funzionalità non correlate (se una classe fa troppe cose, i test per una funzionalità potrebbero fallire a causa di cambiamenti in un'altra)
- *Esempio:* se abbiamo una classe `ReportManager` che genera un report *e* lo invia via email, violiamo SRP. Per testarne la logica di generazione dovremmo anche gestire l'email. Separando le responsabilità in due classi (`ReportGenerator` e `EmailSender` ad es.), possiamo testare la logica di generazione indipendentemente dall'invio email (che potrà essere *mockato* nei test)

Slide 25 – Principio di Inversione delle Dipendenze (DIP)
- Il **Dependency Inversion Principle** (lettera D di SOLID) invita a dipendere da **astrazioni, non da classi concrete**. In pratica, le classi di alto livello non dovrebbero creare direttamente le proprie dipendenze, ma riceverle già pronte (tipicamente tramite interfacce) [27] [28]
- DIP applicato significa progettare moduli in modo che *dichiarino ciò di cui hanno bisogno* (esponendo costruttori che accettano interfacce), anziché *prendere ciò di cui hanno bisogno* (inizializzando oggetti concreti all'interno)
- **Testabilità:** questo principio è alla base del codice testabile. Se una classe dipende da un'interfaccia `IDatabase`, nei test possiamo fornirle un'implementazione finta di `IDatabase` (un *stub* o *mock*), isolando la logica della classe dai dettagli dell'accesso ai dati [29]
- Il DIP spesso è realizzato tramite il meccanismo della **Dependency Injection**, di cui vediamo un esempio a seguire. In C# e .NET, esistono container DI che risolvono automaticamente le dipendenze, ma il concetto chiave resta: dipendere da astrazioni rende il codice più modulare e facilmente sostituibile nei test

Slide 26 – Dependency Injection (Concetto)
- **Dependency Injection (DI)** è la tecnica pratica per applicare il DIP: una classe non crea da sé le proprie dipendenze, ma le riceve dall'esterno (tipicamente tramite il costruttore) [28] . L'"inversione" è che è un codice esterno (il *composer* o il framework) a fornire le istanze necessarie
- **Senza DI:** classi fortemente accoppiate. Esempio: una classe `ReportManager` all'interno del suo metodo chiama `new EmailService()` per inviare email. Ciò la vincola a quella implementazione e rende difficile testarla senza inviare email reali
- **Con DI:** classi flessibili. `ReportManager` dichiara nel costruttore di aver bisogno di un `IEmailService`. Sarà il codice chiamante a passarle un'istanza concreta (che può essere `EmailService` oppure un *mock* nei test). La classe non sa *come* sia implementato `IEmailService`, sa solo usarlo secondo l'interfaccia
- DI favorisce codice **riusabile e modulare**: possiamo cambiare implementazione (es. usare `SmsService` al posto di `EmailService`) senza toccare `ReportManager`. Nei test, possiamo fornire facilmente una finta implementazione per simulare vari scenari

Slide 27 – Esempio senza Dependency Injection
Il seguente esempio mostra una classe senza DI, accoppiata alla sua dipendenza:

```
public class ReportManager {
    public void InviaReport(string testo) {
        // Dipendenza creata internamente (accoppiamento forte)
        EmailService emailService = new EmailService();
        // Uso diretto della dipendenza concreta
        emailService.InviaEmail(testo);
    }
}
```

- `ReportManager` in questo caso **decide autonomamente** come inviare il report (crea un `EmailService`). Per testare `ReportManager` saremmo costretti a inviare veramente un'email o a modificare il codice (non ideale)
- Problemi:
- **Difficile da testare:** non possiamo sostituire facilmente `EmailService` con una versione finta. Il comportamento di invio email non è controllabile nei test (rischia di mandare email reali o va escluso manualmente)

- **Scarsa flessibilità:** se volessimo inviare report via SMS, dovremmo modificare il codice di `ReportManager`. Viola l'apertura/chiusura (OCP) e il DIP: la classe di alto livello dipende da una di basso livello concreta

Slide 28 – Esempio con Dependency Injection
Applichiamo DI refactorando `ReportManager` *per dipendere da un'interfaccia astratta:*

```
public interface INotificatore {
    void Invia(string messaggio);
}
public class EmailService : INotificatore {
    public void Invia(string messaggio) {
        Console.WriteLine("[Email] " + messaggio);
    }
}
// Classe dipendente da astrazione INotificatore
public class ReportManager {
    private readonly INotificatore _notifica;
    public ReportManager(INotificatore servizioNotifica) {
        _notifica = servizioNotifica;
    }
    public void InviaReport(string testo) {
        _notifica.Invia(testo);
    }
}
```

- Ora `ReportManager` non conosce la classe concreta di notifica: usa qualsiasi implementazione di `INotificatore` che gli viene passata. In produzione passeremo un `EmailService`, ma **nei test possiamo passarle un finto**
- Vantaggi:
- Possiamo testare `ReportManager` senza inviare email reali, fornendo nei test un `INotificatore` fittizio (ad es. un mock che registra le chiamate)
- Aggiungere un nuovo modo di notifica (es. SMS) non richiede modifiche a `ReportManager`. Basta creare un'altra classe che implementa `INotificatore` e iniettarla dove serve (aperto all'estensione, chiuso alla modifica)

Slide 29 – Vantaggi della DI per i Test
- **Sostituibilità delle dipendenze:** grazie alla DI, nei test possiamo facilmente **iniettare implementazioni alternative** (stub, fake o mock) al posto di componenti reali onerosi o esterni. Ad esempio, un database può essere sostituito da un repository in-memory durante i test
- **Isolamento:** le classi diventano testabili in isolamento, perché possiamo interrompere le dipendenze vere. Come visto, `ReportManager` può essere testato senza eseguire davvero `EmailService`. Si testa la logica interna senza effetti collaterali esterni [30]
- **Maggiore controllo:** con dipendenze simulate possiamo forzare condizioni difficili da riprodurre (es. far lanciare eccezioni a una dipendenza) e verificare che la classe reagisca correttamente. Questo porta a test più completi e affidabili
- **Design migliorato:** il bisogno di rendere le cose iniettabili spesso spinge verso una progettazione modulare e a definire chiari contratti (interfacce) tra componenti. Il codice risulta più aderente a **DIP/SOLID** e in generale di qualità superiore, non solo più testabile

Slide 30 – Refactoring guidato dai Test

- **Refactoring** = migliorare la struttura interna del codice senza cambiarne il comportamento esterno [31] [7]. Nel ciclo TDD il refactoring è una fase esplicita dopo aver reso verdi i test
- Avere una suite di test automatizzati e affidabili è fondamentale per refactoring sicuri: i test fungono da **regressione automatica**. Se dopo una modifica i test rimangono verdi, abbiamo alta confidenza di non aver rotto funzionalità esistenti
- Esempio dal nostro Stack TDD: abbiamo iniziato con una struttura semplice (una variabile `_soloElemento`) e poi refattorizzato a una lista. I test hanno garantito che il comportamento rimanesse corretto dopo la modifica strutturale
- Durante il refactoring, è buona pratica procedere per passi piccoli: modificare una cosa alla volta ed eseguire subito i test. Se un test fallisce, sapremo subito qual è l'ultima modifica responsabile

Slide 31 – Tecniche di Refactoring Comune

- **Eliminazione delle duplicazioni:** se trovi blocchi di codice ripetuti, estraili in un metodo o classe condivisa. TDD tende a far emergere duplicazioni dopo aver implementato rapidamente soluzioni simili – il refactoring è il momento di unificare. *"Duplication is the root of all evil."*
- **Miglioramento dei nomi:** rinomina metodi, classi e variabili per chiarirne l'intento (ad esempio, un metodo `Calcola()` può diventare `CalcolaTotaleOrdine()`). I test (anch'essi da rinominare se necessario) assicurano che il comportamento resti invariato
- **Suddivisione di funzioni/classes:** se una funzione è troppo lunga o una classe fa troppe cose (violando SRP), refactoring significa **spezzettarla** in unità più piccole e coese. Ad esempio, una funzione con molti `if` /rami logici può essere divisa in funzioni private più specifiche
- **Riordino codice e miglioramento performance:** puoi spostare metodi tra classi (se stanno meglio altrove), semplificare algoritmi, introdurre caching, ecc., purché i test continuino a passare. I test garantiscono che "pulendo" non cambierai il risultato esterno
- **Nota:** durante refactoring, **non aggiungere nuove funzionalità**. Concentrati sul rendere il codice esistente più chiaro e manutenibile. Per nuove feature, torna in modalità Red (nuovi test) e poi Green

Slide 32 – Test Doubles: Stub vs Mock

- Quando si testano unità di codice isolandole dalle dipendenze esterne, si usano oggetti fittizi detti **test double** [32]. Tipi principali:
- **Stub:** oggetto fake con implementazione minima, usato per fornire al codice sotto test dati di ingresso controllati. Lo stub generalmente restituisce valori predefiniti alle chiamate (non contiene logica complessa). Non verifica nulla, serve solo a permettere al test di procedere con input stabili
- **Mock:** oggetto fake più sofisticato che oltre a poter fornire risposte, **registra come viene usato**. Un mock può validare nel test quante volte un certo metodo è stato chiamato, con quali parametri ecc. (verifica del comportamento interno)
- **Esempio:** se stiamo testando una classe `Calcolatore` che somma numeri ottenuti da un servizio esterno `INumeriService`:
- Possiamo usare uno *stub* di `INumeriService` che ritorna sempre una lista fissa di numeri, così da testare il calcolo senza chiamare il vero servizio
- Possiamo usare un *mock* di `INumeriService` per verificare che `Calcolatore` chiami ad esempio il metodo `OttieniNumeri()` esattamente una volta. Il mock ci permette di intercettare e controllare l'interazione con la dipendenza
- In pratica, i framework di mocking in .NET (Moq, NSubstitute, FakeItEasy, ecc.) permettono di creare sia behavior stub (ritorni predefiniti) che veri e propri mock (con aspettative sulle chiamate)

Slide 33 – Introduzione a Moq

- **Moq** è una popolare libreria di mocking per .NET. Consente di creare implementazioni fake di interfacce o classi e di configurarne il comportamento atteso nei test in modo molto semplice
- Con Moq si lavora creando un oggetto `Mock<T>` dove `T` è l'interfaccia (o classe) da simulare. Ad

esempio: `var emailMock = new Mock<IEmailService>();`
- Una volta creato il mock, abbiamo:
- `emailMock.Object` – la **istanza finta** di `IEmailService` da passare al codice sotto test
- `emailMock.Setup(...)` – metodo per definire il comportamento di uno specifico metodo/proprietà quando viene chiamato
- `emailMock.Verify(...)` – metodo per verificare che una certa chiamata sia avvenuta (eventualmente controllando parametri o numero di invocazioni)
- Moq aiuta a scrivere test più **leggibili** e potenti, perché evita di dover creare manualmente classi fake. Con poche linee possiamo dire "quando il metodo X viene chiamato con argomento Y, restituisci Z" oppure "verifica che metodo W sia stato chiamato N volte" 33 34
- Nelle slide seguenti vedremo piccoli esempi di utilizzo di Moq per illustrare questi concetti

Slide 34 – Moq: Simulare un Comportamento (Setup)
Supponiamo di voler testare una classe `CounterService` *che chiede a un repository il valore corrente e restituisce il successivo (incrementandolo di 1).*

```
public interface ICounterRepo {
    int GetCount();
}
public class CounterService {
    private ICounterRepo _repo;
    public CounterService(ICounterRepo repo) { _repo = repo; }
    public int GetNextCount() {
        int current = _repo.GetCount();
        return current + 1;
    }
}
```

- Nel test non vogliamo dipendere da un vero database/repository. Usiamo Moq per creare un **stub** di `ICounterRepo`:

```
var repoMock = new Mock<ICounterRepo>();
repoMock.Setup(r => r.GetCount()).Returns(41);
var service = new CounterService(repoMock.Object);
// Act
int risultato = service.GetNextCount();
// Assert
Assert.Equal(42, risultato);
```

- **Spiegazione:** abbiamo configurato il mock perché, **quando** qualcuno chiama `repoMock.Object.GetCount()`, **ritorni** sempre `41`. Il `CounterService` non sa che è un fake: chiamando `_repo.GetCount()` ottiene 41, quindi `GetNextCount()` restituirà 42. Il test verifica che la logica di `CounterService` funzioni correttamente con il valore simulato dal repository
- In questo scenario Moq ci permette di testare l'incremento senza dover predisporre un database reale con valore 41. Abbiamo **isolato** `CounterService` e controllato totalmente la sua dipendenza

Slide 35 – Moq: Verificare Interazioni (Verify)
Ora consideriamo una classe `Notificatore` *che avvolge un servizio di notifica e vogliamo assicurarci che invii effettivamente i messaggi.*

```
public interface IEmailService {
    void InviaEmail(string msg);
}
public class Notificatore {
    private IEmailService _svc;
    public Notificatore(IEmailService svc) { _svc = svc; }
    public void InviaNotifica(string testo) {
        // ... eventuale logica ...
        _svc.InviaEmail(testo);
    }
}
```

- Nel test, useremo un mock per intercettare la chiamata a `InviaEmail`:

```
var emailMock = new Mock<IEmailService>();
var notif = new Notificatore(emailMock.Object);
// Act
notif.InviaNotifica("Hello");
// Assert: verifichiamo che InviaEmail sia stato chiamato esattamente una
volta con "Hello"
emailMock.Verify(s => s.InviaEmail("Hello"), Times.Once());
```

- Qui non ci interessa il **ritorno** (il metodo è `void`), ma il fatto che il metodo sia stato invocato correttamente. `Verify` ci permette di esprimere questa aspettativa: "sul mock di IEmailService deve essere avvenuta una chiamata a `InviaEmail("Hello")` una volta"
- Se `InviaNotifica` non chiamasse il metodo o lo chiamasse con un testo differente, il nostro test fallirebbe. In questo modo stiamo testando non solo lo stato finale ma anche il **comportamento interno** della classe sotto test (nel rispetto di ciò che è osservabile dall'esterno, ovvero la collaborazione con IEmailService) 35 36
- **Nota:** è bene usare `Verify` solo per interazioni significative. Abusare di verifiche su dettagli di implementazione può rendere i test fragili rispetto a refactoring (vedi slide "Testare il comportamento, non l'implementazione")

Slide 36 – Struttura AAA dei Test
- Una regola pratica per scrivere test chiari è seguire lo schema **AAA: Arrange – Act – Assert** 37
1. **Arrange (Prepara):** imposta la scena del test – inizializza oggetti, imposta valori di input, configura eventuali mock/stub. In questa sezione si crea lo *scenario* su cui effettuare il test
2. **Act (Agisci):** esegui l'azione che vuoi testare – chiama il metodo della classe sotto test, esegui l'operazione con gli input preparati. È **una sola azione** nella maggior parte dei casi (la singola cosa che stiamo testando)
3. **Assert (Verifica):** controlla che i risultati dell'azione siano quelli attesi – qui utilizziamo le asserzioni (`Assert.*`) per confrontare output, stato o interazioni con i valori attesi
- Esempio applicato: nel test dello slide 34:
- Arrange: creazione `repoMock` e configurazione `Returns(41)`, creazione di `CounterService`
- Act: chiamata a `service.GetNextCount()`

- Assert: verifica con `Assert.Equal(42, risultato)`
- Separare visivamente (anche con righe vuote o commenti) queste tre fasi rende il test più leggibile. Chi legge capisce subito qual è il contesto, quale azione viene compiuta e cosa ci si aspetta come risultato. I test scritti in stile AAA risultano auto-esplicativi e **facili da manutere** [38]

Slide 37 – Nominare chiaramente i Test
- Il nome del metodo di test dovrebbe descrivere chiaramente il **comportamento atteso** in uno scenario specifico. Un buon naming funge da documentazione aggiuntiva
- Convenzione comune: *MetodoCondizioneRisultato*. Ad esempio, `Pop_SuStackVuoto_LanciaInvalidOperation` indica che chiamando Pop a certe condizioni (stack vuoto) ci si aspetta un certo risultato (eccezione InvalidOperationException)
- Un altro stile è usare **"Should"**: ad es. `StackVuoto_ShouldThrowExceptionOnPop`. Ciò evidenzia l'aspettativa ("dovrebbe lanciare eccezione")
- Evitare nomi generici come `Test1`, `TestCalculator` – non dicono nulla. Meglio essere espliciti sul caso in esame: es. `Somma_NumeriPositivi_RitornaSommaCorretta`
- Includere nel nome: il metodo o funzionalità sotto test, lo scenario o input particolare, e l'output/comportamento atteso. Questo porta a nomi lunghi ma altamente descrittivi. Nei test la chiarezza è più importante della brevità
- **Esempio pratico:** nella slide 16 abbiamo usato `Pop_SuStackVuoto_LanciaInvalidOperation`. Dal nome, un collega capirebbe subito cosa fa quello test senza leggerne il contenuto dettagliato

Slide 38 – Buone Pratiche nei Test Unitari
- **Test indipendenti:** ogni test deve poter essere eseguito da solo in qualsiasi ordine senza dipendere da effetti di altri test. Evitare di condividere *state* tra test (es: file temporanei, dati static). Se c'è setup comune, utilizzare costruttori della test class o `[Initialize]` appropriati, ma isolare sempre gli effetti
- **Un'asserzione per scenario:** non è obbligatorio avere una sola `Assert` per test, ma tutte le asserzioni di un test dovrebbero riferirsi allo stesso scenario/funzionalità. Se un test fallisce, deve essere immediatamente chiaro quale comportamento (di quella funzione) non regge più. Se un test verifica troppe cose non correlate, meglio suddividerlo
- **Evitare logica complessa nei test:** i test dovrebbero essere lineari e facili da seguire. Non inserire condizioni, loop o calcoli complicati dentro i test per "calcolare" il risultato atteso – rischi bug anche nel codice del test. Meglio **calcolare offline** e mettere il valore letterale atteso nel test, oppure verificare rispetto a un risultato prodotto da una funzione già testata separatamente
- **Non testare banalità/puro boilerplate:** ad esempio, non serve scrivere test per semplici proprietà get/set automatiche o per metodi che sono one-liner triviali (a meno che contengano logica significativa). Il *return* di un campo o una costante non necessitano di test – concentrati su logica con possibili failure. *"Non inseguire il 100% di coverage a scapito del tempo e della qualità dei test."* [39]
- **Manutenere i test come codice di produzione:** se cambi la firma di un metodo o la logica, aggiorna i test pertinenti mantenendo coerenza. Elimina test obsoleti se una funzionalità viene rimossa. I test dovrebbero evolvere insieme al codice – un test rotto non va ignorato, ma sistemato o rimosso se non più valido

Slide 39 – Testare il Comportamento, non l'Implementazione
- Un buon test unitario verifica **ciò che una unità *fa*** (output, effetti osservabili) piuttosto che *come* lo fa internamente [40]. In questo modo, i test restano validi anche se si rifattorizza l'interno della funzione finché il comportamento esterno rimane corretto
- Esempio: per una funzione `CalcolaBonus(punti)` che deve raddoppiare i punti se > 100, un test di comportamento controlla che `CalcolaBonus(150)` dia `300`. Un test sbagliato di implementazione potrebbe verificare "dopo aver chiamato CalcolaBonus, viene invocato il metodo X e Y" – dettagli interni che potrebbero cambiare nel refactoring

- **Evita over-specification:** se nei test con mock verifichi troppe interazioni interne (es: "metodo A chiama B poi C in quest'ordine"), stai legando il test all'implementazione corrente. Un refactoring che cambia l'ordine delle chiamate (ma non il risultato finale) farebbe fallire il test inutilmente [41]
- Focus sui **risultati attesi**: per input X, l'output deve essere Y, o deve avvenire l'azione Z. Se la logica interna cambia ma questi risultati esterni restano gli stessi, i test dovrebbero continuare a passare. Questo dà libertà allo sviluppatore di migliorare il codice senza dover riscrivere i test ogni volta
- Un approccio per garantire ciò è scrivere molti test come *black box*: tratti la classe sotto test come una scatola nera, dandole input e controllando output/effetti pubblici. Solo quando necessario (ad es. interazioni con dipendenze), usa i *mock* ma con parsimonia, verificando solo ciò che è realmente rilevante per il comportamento osservabile di alto livello

Slide 40 – Organizzare il Codice di Test
- **Struttura dei file:** mantieni una convenzione chiara. Spesso si crea una classe di test per ciascuna classe di produzione (es. test per `Calculator` in `CalculatorTests`). In alternativa, si possono raggruppare test per funzionalità (es. `Calcolatrice_SommaShould` per i test del metodo Somma). L'importante è trovare facilmente dove sono i test di una certa unità
- **Namespace paralleli:** di solito il progetto di test rispecchia la gerarchia del progetto principale. Esempio: i test per `MyApp.Logica.Ordini.Ordine` potrebbero stare in `MyApp.Tests.Logica.Ordini.OrdineTests`. Ciò facilita navigazione e mantenimento
- **Setup comune:** se molti test richiedono un certo oggetto inizializzato (es. un `Service` con certi stub), valuta di usare un costruttore nella classe di test o `[SetUp]` (in xUnit puoi usare il costruttore della test class) per evitare ripetizioni. Ma attenzione a non introdurre dipendenze implicite tra test attraverso lo state condiviso del setup – mantieni il setup idempotente e indipendente per ogni test
- **Esecuzione parallela:** xUnit esegue di default i test in parallelo in classi diverse. Assicurati che i test non si ostacolino (es: non usare la *stessa risorsa esterna* – file, porta di rete – in test paralleli). Puoi disabilitare il parallelismo se necessario con attributi `[Collection]`, ma meglio progettare test thread-safe quando possibile
- **Nomenclatura uniforme:** segui lo stesso stile di naming e struttura in tutto il suite di test. Ciò rende immediata la comprensione. Ad esempio, se usi il formato `Metodo_StatoAtteso_Risultato`, usalo ovunque per coerenza. Coerenza = professionalità e manutenibilità

Slide 41 – Tipi di Test Automatizzati
- I test unitari (come quelli scritti in TDD) sono alla base della piramide dei test. Sono **molti**, molto veloci, coprono piccole unità di codice isolatamente. Danno feedback immediato sul corretto funzionamento di funzioni/metodi in ogni condizione prevista [42]
- Al livello superiore ci sono i **test di integrazione**: verificano l'interazione di più componenti tra loro (ad esempio accesso reale a DB, chiamate HTTP reali a un servizio). Sono meno numerosi e più lenti degli unit test, ma essenziali per coprire scenari in cui conta che i moduli lavorino bene assieme
- Al top ci sono i **test end-to-end (E2E)** o **test funzionali**: simulano scenari completi dal punto di vista dell'utente o del sistema nel suo insieme (es: usare l'applicazione completa, chiamare API reali con un database reale). Sono pochi e molto lenti, ma validano che l'intero sistema soddisfi requisiti business
- **Piramide dei test:** molti unit test alla base, un numero moderato di test di integrazione al centro, pochi test E2E in cima [43] [44]. Questo equilibrio garantisce una buona copertura con velocità: usiamo test unitari per la maggior parte delle verifiche e solo quando serve testiamo integrazioni e flussi end-to-end
- In un processo completo di QA, TDD copre la parte dei test unitari e parte degli integration test (se li scriviamo in anticipo). Non sostituisce comunque la necessità di test più alti: ad esempio, dopo aver fatto TDD sul dominio, dovremo comunque fare test di sistema per verificare configurazioni, pipeline reali, ecc.

Slide 42 – Automazione dei Test e CI

- Integrare l'esecuzione dei test automatici nel processo di sviluppo quotidiano e di build è fondamentale. Idealmente, ogni volta che si apporta una modifica significativa al codice, **i test vengono eseguiti automaticamente** (ad es. tramite una pipeline di Continuous Integration)
- **Continuous Integration (CI):** ogni commit sul repository triggera una build automatica che include l'esecuzione di tutti i test unitari (e spesso anche integration). Se qualche test fallisce, la pipeline segnala immediatamente il problema al team [45]
- I sviluppatori ricevono così feedback continuo: se il codice introdotto rompe qualcosa, lo sanno subito. Questo aiuta a **catturare regressioni** presto, quando è ancora fresco cosa è stato cambiato [45]
- **Local vs Remote:** è buona pratica eseguire tutti i test in locale prima di fare push. Inoltre, attivare la pipeline CI sul server garantisce che i test girino anche in un ambiente "pulito" e identifichino problemi ambientali o di dipendenze mancanti
- **Coverage nella CI:** molte pipeline calcolano anche la percentuale di copertura del codice dai test. Pur non dovendo inseguire ciecamente il 100%, un trend di coverage decrescente può indicare che nuove parti di codice non stanno venendo testate adeguatamente

Slide 43 – Misurare la Copertura del Codice

- La **code coverage** indica la percentuale di righe (o rami) di codice eseguiti almeno una volta dai test. È uno strumento utile per individuare zone non coperte dai test, ma va interpretato con attenzione
- **Utilità:** se un file mostra 0% di copertura e contiene logica, probabilmente manca test per quella logica. Aumentare la copertura riduce la possibilità che parti di codice non verificate contengano bug sconosciuti
- **Limiti:** un alto numero di coverage non garantisce automaticamente assenza di bug. È possibile scrivere test superficiali che esercitano linee di codice senza davvero verificare risultati significativi. Meglio **poche asserzioni mirate** che tante righe "toccate" ma non validate
- Non ossessionarsi col 100%: come detto, non ha senso testare codice triviale o getter/setter. Kent Beck stesso afferma che è accettabile **non testare il codice banale** [39] . Concentrarsi invece sul coprire tutte le logiche condizionali e i percorsi di calcolo complessi
- In sintesi: usiamo la coverage come **indicatore** (se è molto bassa c'è da aggiungere test), ma non come obiettivo fine a se stesso. La qualità dei test conta più della quantità di righe coperte

Slide 44 – Sfide e Limiti del TDD (I)

- **Investimento iniziale:** adottare TDD richiede tempo e disciplina. All'inizio, lo sviluppo può sembrare più lento perché si scrive del codice in più (i test) prima di vedere funzionalità funzionanti. Serve pratica per diventare veloci nel ciclo Red-Green-Refactor
- **Non è una bacchetta magica:** TDD **non garantisce** da solo che il design sia perfetto o che non ci saranno bug. Se applicato correttamente aiuta moltissimo (code più semplice, bug scoperti presto), ma non risolve problemi di analisi requisiti o di progettazione ad alto livello [46] . Rimane uno strumento tra tanti nel processo di sviluppo agile
- **Applicabilità variabile:** TDD funziona benissimo per logica di business, algoritmi, codice dove gli output sono prevedibili. In alcuni contesti è meno immediato: GUI/UX (dove test automatici sul layout sono complessi), progetti di data science/prototipi dove gli obiettivi possono essere fluidi. In questi casi, si può adottare un approccio misto (testare a posteriori o usare BDD per specifiche di alto livello)
- **Curve di apprendimento del team:** se il team non è abituato ai test automatizzati, può esserci resistenza. TDD richiede un cambio di mentalità ("prima il test, poi il codice") che va assimilato. Pair programming o mentoring possono aiutare nella transizione

Slide 45 – Sfide e Limiti del TDD (II)

- **Test fragili dovuti a over-specification:** se i test sono scritti controllando dettagli interni o scritti in modo troppo rigido, un refactoring innocuo può farne fallire molti. Questo può scoraggiare dal refactor e portare a test disallineati. È importante scrivere test flessibili concentrati sul comportamento

osservabile, come discusso [41]
- **Manutenzione dei test:** la suite di test va mantenuta come il codice di produzione. Cambi di requisiti e refactoring possono richiedere di aggiornare i test. Un antipattern è "commentare test rotti" invece di sistemarli – pratica che erode il valore della suite. Serve tempo dedicato anche a rivedere e pulire i test nel corso del progetto
- **Falsa sicurezza:** test passanti al 100% non significano che l'app non abbia bug. I test coprono quello che lo sviluppatore prevede; bug dovuti a requisiti fraintesi o casi non pensati possono sfuggire. Inoltre i test unitari non catturano problemi di integrazione tra componenti o con l'ambiente (es. errori di configurazione). Per questo è importante affiancare al TDD altri livelli di test (integrazione, UAT) e la valida analisi dei requisiti
- **Sforzo su test di legacy code:** applicare TDD su codice legacy non coperto da test è difficile: idealmente si dovrebbero introdurre test **a caratterizzazione** prima di modificarlo, ma scrivere test su codice non progettato per essere testabile può essere oneroso. In tali casi, si può procedere con refactoring incrementale per inserire punti di *seam* (dove poter testare) e aggiungere gradualmente test man mano che si ristruttura il codice esistente

Slide 46 – Oltre TDD: BDD e ATDD
- **Behavior-Driven Development (BDD):** è un'evoluzione del TDD che enfatizza la collaborazione e l'uso di un linguaggio condiviso per descrivere il comportamento del software [47]. In BDD si scrivono specifiche di comportamento in termini di **Given-When-Then** (Dato un contesto, Quando accade X, Allora deve succedere Y) spesso usando linguaggio naturale semi-strutturato (es. Gherkin in Cucumber)
- Il BDD sostituisce i "test" con "esempi" comprensibili anche ai non programmatori (analisti, QA, business). Esempio BDD: *Dato un conto con saldo 100€, quando prelevo 30€, allora il saldo finale deve essere 70€*. Queste specifiche possono poi essere automatizzate con test sottostanti
- **Acceptance TDD (ATDD):** simile al BDD, coinvolge scrivere prima i test di accettazione (alto livello, spesso end-to-end) che definiscono se una funzionalità è completata dal punto di vista utente. Solo dopo si implementa la funzionalità finché tali test non passano. ATDD garantisce che lo sviluppo sia guidato dai requisiti utente concreti fin dall'inizio
- **Relazione con TDD classico:** BDD/ATDD si focalizzano sul *cosa* deve fare il sistema più che sul *come* a livello unità. Spesso si usano in combinazione: i test di accettazione BDD guidano lo sviluppo macro, e il TDD unitario guida lo sviluppo delle singole componenti interne per soddisfare quelle specifiche
- In .NET, esistono framework come **SpecFlow** che permettono di definire scenari BDD in linguaggio naturale e collegarli a codice di test C#. Questo può aumentare la comprensione condivisa, ma introduce un ulteriore strato. È utile valutarlo quando serve allineare fortemente team tecnici e non tecnici sulle specifiche

Slide 47 – Risorse Consigliate
- **Libro:** *Test Driven Development: By Example* – Kent Beck. Il testo fondamentale sul TDD scritto dal suo ideatore, ricco di esempi pratici passo passo
- **Libro:** *The Art of Unit Testing* – Roy Osherove. Guida pratica ai test unitari in .NET, copre anche argomenti come mocking, design for testability e organizzazione dei test
- **Documentazione xUnit.net:** Guida ufficiale e esempi d'uso di xUnit sul sito [48] [49]
- **Libreria Moq:** Repositorio GitHub e Wiki di Moq (descrizione di tutti i metodi `Setup`, `Verify`, ecc.) per approfondire l'API e scenari avanzati
- **Articolo:** *"The Practical Test Pyramid"* di Martin Fowler [50] [51]. Approfondisce la strategia di bilanciamento tra test unitari, integrazione e end-to-end
- **Community & Blog:** seguire blog e conferenze (.NET) sul testing – es. *JetBrains .NET blog*, *Pluralsight courses on Unit Testing*, e la sezione di documentazione Microsoft "Unit testing best practices" [52] [53] – per rimanere aggiornati su strumenti e tecniche emergenti

Slide 48 – Conclusioni
- Il Test-Driven Development è più di una tecnica di testing: è una metodologia di **design del codice**. Scrivendo i test prima, pensiamo a *come* vogliamo poter usare il codice (API) e quali risultati deve produrre, ancor prima di scrivere l'implementazione [54]. Questo porta a soluzioni più semplici e mirate ai requisiti
- Adottare TDD in C# consente di sfruttare appieno il potente ecosistema .NET: framework come xUnit e Moq rendono relativamente agevole integrare i test nel flusso di sviluppo quotidiano
- Abbiamo visto come TDD incoraggi il rispetto dei principi SOLID (SRP, DIP...) migliorando la struttura del codice. Il risultato tipico di TDD è un codice meno accoppiato e più robusto, con benefici a lungo termine sulla manutenibilità
- La chiave del successo con TDD è la **costanza**: applicarlo regolarmente fin dalle prime fasi di un progetto. I primi test guideranno la prima implementazione, e la suite crescerà insieme al codice fungendo da scudo contro regressioni future
- **Prossimi passi:** provare ad applicare TDD in un piccolo progetto o *kata* (esercizio di coding) per acquisire confidenza. Ad esempio, sviluppare con TDD un convertitore Roman Numerals, o il gioco del FizzBuzz, o altre kata note. L'esperienza diretta aiuterà a interiorizzare il ritmo Red-Green-Refactor
- In conclusione, TDD offre un modo affidabile per aumentare la qualità del software e la confidenza nello sviluppo: *"Clean code that works"* è il motto. All'inizio può sembrare controintuitivo scrivere test prima del codice, ma una volta sperimentati i benefici (meno bug, design pulito, deployment più sereni) difficilmente si torna indietro! [14]

Slide 49 – Domande e Discussione
- Hai un dubbio su come applicare TDD in un caso reale del tuo progetto? Condividilo!
- Domande sui framework visti (xUnit, Moq) o su come gestire situazioni specifiche (es. test di codice multi-thread, test di API esterne)?
- **Condivisione di esperienze:** qualcuno di voi ha già utilizzato TDD o parti di queste pratiche? Com'è andata, quali difficoltà iniziali?
- Sentitevi liberi di proporre un piccolo esercizio da ragionare insieme in ottica TDD, oppure di chiedere chiarimenti su qualsiasi concetto presentato

Slide 50 – Grazie!
- Grazie per l'attenzione. Spero che questa lezione vi abbia fornito spunti pratici per migliorare il vostro workflow con TDD
- *"Scrivere il codice due volte (una nei test e una nell'implementazione) può sembrare uno sforzo doppio, ma in realtà vi risparmierà di scriverlo dieci volte durante il debugging."* – cit. proverbiale nel mondo TDD
- **Contatti:** per approfondimenti o domande post-lezione, potete contattarmi via email o sul forum interno. Buon coding e buon testing a tutti!

---

[1] [2] [3] [6] [7] [14] [31] Test-driven development - Wikipedia
https://en.wikipedia.org/wiki/Test-driven_development

[4] [5] [8] [9] [54] What is TDD (Test Driven Development)
https://testguild.com/what-is-tdd/

[10] [11] [12] [13] [21] [22] [23] [24] [25] [45] What is Test Driven Development (TDD) ? | BrowserStack
https://www.browserstack.com/guide/what-is-test-driven-development

[15] Discover xUnit: The Essential .NET Unit Testing Framework
https://www.clariontech.com/blog/why-should-you-use-xunit-a-unit-testing-framework-for-.net

16 17 18 19 20 48 49 Unit testing C# code in .NET using dotnet test and xUnit - .NET | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-csharp-with-xunit

26 Single Responsibility Principle (SRP) | by Shashi Kant | Medium
https://medium.com/@shashikantrbl123/single-responsibility-principle-srp-4700d3c668aa

27 28 29 30 C# Propedeutico 3.pdf
file://file-Rf89AxzfnPgkvVfuZFRQ7t

32 37 38 39 40 43 46 50 51 The Practical Test Pyramid
https://martinfowler.com/articles/practical-test-pyramid.html

33 34 35 36 41 What is Mocking? Mocking in .NET Explained With Examples
https://www.freecodecamp.org/news/explore-mocking-in-net/

42 The test pyramid: A complete guide - Qase
https://qase.io/blog/test-pyramid/

44 Why the Test Pyramid Still Matters for Engineering Teams - QAlified
https://qalified.com/blog/test-pyramid-for-engineering-teams/

47 What is BDD (Behavior Driven Development)? | Agile Alliance
https://agilealliance.org/glossary/bdd/

52 Unit Test Mocking: What You Need to Know - HyperTest
https://www.hypertest.co/unit-testing/unit-test-mocking

53 Best practices for writing unit tests - .NET - Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices