Test di Unità con xUnit in .NET Core

Introduzione: perché i test di unità e TDD (Test-Driven Development)

- **Importanza dei test di unità**: forniscono feedback rapido sul comportamento di piccole parti del codice (funzioni/classi) in isolamento. Aiutano a:
  - **Prevenire regressioni** e aumentare la confidenza durante i refactor;
  - **Guidare il design** verso componenti più **modulari e testabili** (meno accoppiamento, dipendenze esplicite);
  - Servire come **documentazione eseguibile** degli scenari supportati;
  - **Localizzare i difetti** precocemente con tempi di esecuzione minimi.
- **Test-Driven Development (TDD)**: pratica in cui si scrive prima un test che fallisce, poi il codice minimo per farlo passare, infine si refactora mantenendo verdi i test.
  - Ciclo **Red** → **Green** → **Refactor**:
    - Red: scrivi un test che descrive il comportamento desiderato (fallisce);
    - Green: implementa la soluzione più semplice per far passare il test;
    - Refactor: migliora il design mantenendo i test verdi.
  - Benefici: design più pulito, **copertura mirata** sugli casi rilevanti, **ritmo di sviluppo sostenibile** grazie al feedback continuo.
- **Altri tipi di test** (oltre ai unitari), ciascuno utile in momenti diversi:
  - **Integrazione**: verificano l'interazione fra componenti, DB, file, API esterne;
  - **End-to-End/Functional/UI**: simulano i flussi utente sull'app completa;
  - **Performance/Load/Stress**: misurano tempi, throughput e resilienza sotto carico;
  - **Security**: cercano vulnerabilità e problemi di configurazione;
  - **Property-based testing**: validano invarianti su molti input (es. QuickCheck/FsCheck);
  - **Mutation testing**: valuta l'efficacia dei test introducendo mutazioni nel codice.

Suggerimenti pratici TDD in .NET:

- Mantieni i test **veloci e deterministici** (niente rete/DB: usa fakes/stubs/mocks);
- Progetta con **Dependency Injection** per isolare le dipendenze;
- Usa `[Theory]` per coprire varianti di input con un solo test;
- Nomina i test per comportamento: `Metodo_Condizione_RisultatoAtteso`;
- Evita test fragili che dipendono da dettagli di implementazione o dall'ordine di esecuzione.

1. Creazione di un progetto di test xUnit

1. Apri il terminale nella root della soluzione.
2. Crea la cartella per i test (opzionale, ma consigliato):

```
mkdir Cards.Tests
```

3. Crea il progetto di test xUnit:

```
dotnet new xunit -n Cards.Tests
```

4. Aggiungi il riferimento al progetto da testare:

```
dotnet add Cards.Tests/Cards.Tests.csproj reference Cards/Cards.csproj
```

2. Strutturazione dei test: teoria e best practice

- **Organizzazione**: crea una classe di test per ogni classe da testare (es. MazzoTests per Mazzo).
- **Nomenclatura**:
  - I metodi di test devono essere pubblici, restituire void o Task e avere l'attributo `[Fact]` (test singolo) o `[Theory]` (test parametrizzato).
  - Usa nomi descrittivi: `MetodoDaTestare_Condizione_RisultatoAtteso`.
- **Indipendenza**: ogni test deve essere indipendente dagli altri.
- **Setup**: usa il costruttore della classe di test o `[Setup]` per preparare lo stato iniziale.
- **Assert**: verifica il risultato atteso con `Assert.Equal`, `Assert.True`, ecc.
- **Copertura**: testa sia i casi normali che quelli limite/errore.
- **Dati di test**: usa dati inline o helper, evita dipendenze esterne (file, DB).
- **Pulizia**: elimina risorse temporanee nel teardown (se necessario).

Esempio di test

```
using Xunit;
using Cards;

public class MazzoTests
{
    [Fact]
    public void Mazzo_NuovoMazzo_Ha52Carte()
    {
        var mazzo = new Mazzo();
        Assert.Equal(52, mazzo.Carte.Count);
    }
}
```

3. Esecuzione dei test da riga di comando

1. Posizionati nella root della soluzione.

2. Esegui i test:

```
dotnet test Cards.Tests/Cards.Tests.csproj
```

oppure, per tutti i progetti di test:

```
dotnet test
```

3. Visualizza il risultato nel terminale.

4. Best practice aggiuntive

- Integra i test nella pipeline CI/CD (GitHub Actions, Azure DevOps, ecc.).
- Mantieni i test aggiornati e fallibili (un test che non può fallire non serve).
- Scrivi test chiari e leggibili.
- Usa `[Theory]` e `[InlineData]` per testare più casi con lo stesso metodo.
- Esegui i test frequentemente durante lo sviluppo.

5. Approfondimento: `[Theory]` e `[InlineData]`

Quando un comportamento deve essere verificato su più input, usa `[Theory]` con `[InlineData]` (o `[MemberData]`/`[ClassData]` per dataset complessi).

- **`[Fact]`**: un singolo caso di test.
- **`[Theory]`**: lo stesso test ripetuto per più valori.
- **`[InlineData]`**: valori inline direttamente sull'attributo (semplici tipi). Per dataset ricchi, preferisci **`[MemberData]`** o **`[ClassData]`**.

Esempio pratico: test parametrizzati di `ValutatorePoker` (Scala Reale)

Obiettivo: dimostrare come usare `[Theory]` + `[InlineData]` per verificare che una mano sia riconosciuta come `ScalaReale` e prevenire falsi positivi (es. scala colore con asso basso).

Nota: adatta la conversione da stringhe a `Carta` in base al tuo modello (costruttori, enum `Valore`/`Seme`, ecc.).

```
using Xunit;
using System.Collections.Generic;
using Cards;

public class ValutatorePokerTheoryTests
{
   // Ogni InlineData passa: array di carte (rank+suit) e punteggio atteso
   // Convenzione: rank = A,K,Q,J,10,9..2; suit = H (cuori), D (quadri), C
(fiori), S (picche)

   [Theory]
   [InlineData(new[] { "10H", "JH", "QH", "KH", "AH" }, PunteggioPoker.ScalaReale,
"Scala Reale di cuori")]
   [InlineData(new[] { "AH", "2H", "3H", "4H", "5H" }, PunteggioPoker.ScalaColore,
"Scala colore con asso basso (non reale)")]
   [InlineData(new[] { "10H", "JH", "QH", "KH", "AD" }, PunteggioPoker.Scala,
"Scala non di colore (non reale)")]
   public void ValutaMano_ScenariScala(string[] codiciCarte, PunteggioPoker
```

```
punteggioAtteso, string motivo)
    {
        // Arrange: converte stringhe in List<Carta>
        var mano = CreaManoDaCodici(codiciCarte);

        // Act
        var (punteggio, descrizione) = ValutatorePoker.ValutaMano(mano);

        // Assert
        Assert.Equal(punteggioAtteso, punteggio);
    }

    private static List<Carta> CreaManoDaCodici(string[] codici)
    {
        var mano = new List<Carta>();
        foreach (var codice in codici)
        {
            // Parsing semplice: split rank e suit (es. "10H", "AH")
            var suit = codice[^1]; // ultima lettera
            var rankStr = codice.Substring(0, codice.Length - 1);

            int valoreNumerico = rankStr switch
            {
                "A" => 14,
                "K" => 13,
                "Q" => 12,
                "J" => 11,
                "10" => 10,
                "9" => 9,
                "8" => 8,
                "7" => 7,
                "6" => 6,
                "5" => 5,
                "4" => 4,
                "3" => 3,
                "2" => 2,
                _ => throw new System.ArgumentException($"Rank non valido: {rankStr}")
            };

            // TODO: Adatta questa creazione a come è definita `Carta` nel tuo
progetto.
            // Esempio generico con inizializzatori di proprietà:
            var carta = new Carta
            {
                // Valore = (mappa rankStr all'enum Valore, se presente)
                // ValoreNumerico = valoreNumerico (se è una proprietà impostabile)
                // Seme = mappa suit a enum Seme ('H','D','C','S')
            };

            mano.Add(carta);
        }

        return mano;
```

```
    }
}
```

Best practice con Theory

- Metti la logica di parsing/conversione in metodi helper per evitare duplicazione.
- Copri scenari positivi, negativi e edge case (es. asso basso). Nel poker: verifica che `AH-2H-3H-4H-5H` sia `ScalaColore` ma non `ScalaReale`.
- Usa `[MemberData]` o `[ClassData]` quando i parametri diventano complessi (es. oggetti completi `Carta`), mantenendo i test leggibili.
- Evita InlineData troppo lunghi: in tal caso sposta i dati in `MemberData`.

Esecuzione mirata dei test

- Solo i test della classe: `dotnet test --filter FullyQualifiedName~ValutatorePokerTheoryTests`
- Solo un metodo: `dotnet test --filter "FullyQualifiedName=Cards.Tests.ValutatorePokerTheoryTests.ValutaMano_ScenariScala"`