Gestione degli esercizi delle prove svolte durante la professione di docente

1 Introduzione 4

1.1 Informazioni sul progetto 4

1.2 Abstract 4

1.3 Scopo 4

Analisi 5

1.4 Analisi del dominio 5

1.5 Analisi e specifica dei requisiti 5

1.6 Use case 7

1.7 Pianificazione 8

1.8 Analisi dei mezzi 9

1.8.1 Software 9

1.8.2 Hardware 9

2 Progettazione 10

2.1 Design dell’architettura del sistema 10

2.2 Design dei dati e database 12

2.3 Design delle interfacce 13

2.3.1 Schermata principale 13

2.3.2 Impostazioni di base 14

2.3.3 Esercizi 15

2.3.4 Creazione prova 16

2.4 Design procedurale 17

3 Implementazione 18

3.1 Creazione progetto 18

3.2 Libreria di Classi 19

3.2.1 Models 19

3.2.1.1 BaseEntity 19

3.2.1.2 Classe 19

3.2.1.3 Modulo 19

3.2.1.4 Tematica 19

3.2.1.5 Anno 20

3.2.1.6 Esercizio 20

3.2.1.7 EsercizioProva 20

3.2.1.8 Prova 20

3.2.2 Services 21

3.2.2.1 IDataRepository 21

3.2.2.2 DbDataRepository 21

3.2.2.3 IClasseRepository 22

3.2.2.4 ClasseDbRepository 22

3.2.2.5 IModuloRepository 22

3.2.2.6 ModuloDbRepository 22

3.2.2.7 ITematicaRepository 22

3.2.2.8 TematicaDbRepository 22

3.2.2.9 IEsercizioRepository 23

3.2.2.10 EsercizioDbRepository 23

3.2.2.11 IProvaRepository 23

3.2.2.12 ProvaDbRepository 23

3.2.3 AppDbContext 24

3.3 App WPF 25

3.3.1 ViewModels 25

3.3.1.1 MainViewModel 26

3.3.1.2 BenvenutoViewModel 27

3.3.1.3 AboutViewModel 27

3.3.1.4 GuidaViewModel 27

3.3.1.5 EsercizioListViewModel 27

3.3.1.6 ProvaListViewModel 27

3.3.1.7 ImpostazioniBaseViewModel 27

3.3.1.8 EsercizioViewModel 27

3.3.1.9 ProvaViewModel 27

3.3.2 Views 28

3.3.2.1 MainView 28

3.3.2.2 BenvenutoView 28

3.3.2.3 AboutView 29

3.3.2.4 GuidaView 29

3.3.2.5 EsercizioListView 29

3.3.2.6 ProvaListView 29

3.3.2.7 ImpostazioniBaseView 29

3.3.2.8 EsercizioView 29

3.3.2.9 ProvaView 29

4 Test 30

4.1 Protocollo di test 30

4.2 Risultati test 30

4.3 Mancanze/limitazioni conosciute 30

5 Consuntivo 31

6 Conclusioni 32

6.1 Sviluppi futuri 32

6.2 Considerazioni personali 32

7 Bibliografia 32

7.1 Sitografia 32

8 Allegati 32

# Introduzione

## Informazioni sul progetto

* Allievo: Gabriele Alessi   
  Superiore professionale: Ugo Bernasconi
* Scuola d’Arti e Mestieri di Trevano, Sezione informatica, Classe 4, Progetti individuali
* Data inizio: 03.09.2019   
  Data fine: 20.12.2019

## Abstract

The teachers of the Scuola d’Arti e Mestieri di Trevano use to create the exercises and the tests without any type of help from a tool or system. This project consists of creating a computer program that allows you to create and manage the exercises and the tests quickly and easily. In detail, the program works so you can set up the basic info (subjects, classes and themes) to then use them to create the exercises.   
When you create an exercise, you have a global vision of the basic settings and then you can define the text and eventually insert an image. Finally, you can create the document by entering the fundamental fields (title, date, class and subject) and selecting the related exercises.   
The application is entirely developed in C# MVVM with Visual Studio 2019 and for the data storage, SQLite is used.

## Scopo

Il progetto consiste nel sviluppare una piccola applicazione che gestisca e crei gli esercizi delle prove per poi prepararne il documento. Il programma deve funzionare in modo che si possano inserire delle informazioni di base: definizione di moduli, tematiche e classi.   
In seguito si gestiscono i veri e propri esercizi, in cui si ha una visione generale delle impostazioni di base al fine di iniziare a inserire i dettagli dell’esercizio (titolo, testo, immagine, …).   
Infine si passa alla creazione del documento, dove vanno definiti i campi fondamentali e si selezionano gli esercizi da inserire nella prova.

Il progetto ha anche uno scopo scolastico, cioè la preparazione al lavoro finale LPI che si svolge alla fine dell’anno per l’ottenimento dell’AFC.

## Analisi

## Analisi del dominio

Fino a ora i professori gestivano e creavano gli esercizi e le prove a mano, quindi l’obiettivo è quello di sviluppare un programma che agevoli appunto la gestione degli esercizi. Il prodotto è leggero e semplice e veloce da usare, quindi può funzionare senza problemi sui computer dei docenti della SAMT.   
Attualmente non sembra esistere una soluzione simile, dunque il sistema funzionerà in modo che sia user-friendly e che più docenti potranno usarlo.

## Analisi e specifica dei requisiti

|  |  |
| --- | --- |
| **ID: REQ-001** | |
| **Nome** | Realizzare un programma che gestisca gli esercizi |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | - |
| **Sotto requisiti** | |
| **001** | È necessario poter inserire delle informazioni di base (REQ-002) |
| **002** | Maschera di inserimento delle informazioni dell’esercizio (REQ-003) |
| **003** | Creazione delle prove (REQ-004) |

|  |  |
| --- | --- |
| **ID: REQ-002** | |
| **Nome** | Impostazioni di base |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | Il programma deve essere strutturato in modo che si possano definire delle impostazioni di base |
| **Sotto requisiti** | |
| **001** | Definizione dei moduli |
| **002** | Definizione delle tematiche (con una sequenza cronologica) |
| **003** | Definizione delle classi |

|  |  |
| --- | --- |
| **ID: REQ-003** | |
| **Nome** | Definizione degli esercizi |
| **Priorità** | 1 |
| **Versione** | 1.1 |
| **Note** | Si devono poter vedere le informazioni principali per poi creare gli esercizi |
| **Sotto requisiti** | |
| **001** | Impostazione informazioni di base |
| **002** | Testo esercizio |
| **003** | Immagine esercizio (Drag & Drop per semplificare l’UI) |

|  |  |
| --- | --- |
| **ID: REQ-004** | |
| **Nome** | Creazione prove |
| **Priorità** | 1 |
| **Versione** | 1.1 |
| **Note** | Deve essere preparato un documento con i vari esercizi |
| **Sotto requisiti** | |
| **001** | Definizione campi fondamentali |
| **002** | Definizione esercizi (Drag & Drop per semplificare l’UI) |
| **003** | Preparazione documento |

## Use case



## Pianificazione

La pianificazione del progetto è stata effettuata mediante la realizzazione di un diagramma di Gantt.

## Analisi dei mezzi

### Software

I software utilizzati per la realizzazione di questo progetto sono:

* Microsoft Word 2016   
  Realizzazione della documentazione.
* GitHub Desktop 2.2.3  
  Gestione del sistema di versioning del progetto.
* Visual Studio Code 1.40  
  Realizzazione dei diari giornalieri e gestione dei vari documenti di testo.
* Microsoft Visual Studio Community 2019  
  Sviluppo del programma.
* Microsoft Project 2016  
  Realizzazione del diagramma di Gantt.
* Microsoft Visio 2016   
  Realizzazione di vari schemi e diagrammi.

### Hardware

Personal Computer:

* HP Envy Notebook
* Intel© Core™ i7-6500U @ 2.50GHz
* 16GB
* Windows 10 Home 64bit
* Intel® HD Graphics 520

# Progettazione

## Design dell’architettura del sistema

In questo capitolo viene spiegata la struttura generale del sistema e in particolare l’architettura del progetto. Il programma è sviluppato in C# utilizzando il pattern MVVM, ma ci sono altri componenti utili per semplificare il lavoro che sono spiegati anche nel capitolo di implementazione.

La struttura del progetto parte da una Soluzione Vuota che contiene un progetto App WPF il quale gestisce l’intero sistema di dati e interfacce.



Per creare questa struttura mi sono valso dell’aiuto del superiore professionale che mi ha consegnato un’estensione che crea automaticamente l’oggetto Model, View, e ViewModel definito.



Per fare ciò ho anche usato il pacchetto DevExpressMvvm che è un insieme di strumenti che agevola il lavoro se si utilizza appunto il pattern MVVM. Inoltre esso lavora insieme all’estensione SAMT, quindi è molto semplice creare la struttura del progetto e gestire il sistema.



Infine nella cartella Services vengono gestiti i DataRepository, cioè tutto ciò che riguarda la struttura dei dati.

In concreto ci saranno delle interfacce e delle classi che impostano le azioni che verranno sui dati (insert, delete, update).



Per il database verrà usato SQLite visto che funziona in modo facile e utilizza semplicemente un file per la memorizzazione dei dati.

## Design dei dati e database

L’immagine seguente rappresenta la progettazione del database del sistema. In generale non è un diagramma molto articolato quindi da esso si può più o meno capire anche la struttura del sistema.



Le entità come *tematica*, *classe*, *anno* e *modulo* presenteranno bene o male sempre gli stessi valori in quanto rappresentano le impostazioni di base. Dopodiché ci sono le entità che simboleggiano gli esercizi e le prove, le quali sono relazionate tramite l’*anno* e il *modulo*. L’oggetto *esercizio* *prova* funge da collegamento tra *prova* e *esercizio* in modo che sia più facile gestire gli esercizi che si trovano nelle prove.

## Design delle interfacce

In questa sezione vengono mostrate e descritte le interfacce dell’applicazione con cui l’utente interagisce.

### Schermata principale

La prima schermata è molto semplice e presenta solamente tre pulsanti che rappresentano le tre azioni principali del programma.



Il menu ha due opzioni e si estendono in questo modo:

* File
  + Esercizi  
    Mostra una schermata con lista degli esercizi salvati nel database.
  + Prove  
    Mostra una schermata con lista delle prove salvati nel database.
  + Opzioni  
    Permette di configurare diverse opzioni come la posizione dei salvataggi.
  + Esci  
    Chiusura dell’applicazione.
* Info
  + Guida  
    Mostra una schermata con una breve guida sul prodotto.
  + About  
    Mostra i diritti d’autore e altre informazioni riguardanti il prodotto (versione, licenza, …).

I tre pulsanti apriranno una nuova schermata in base alla relativa scelta e non sarà possibile uscire da quella schermata senza l’uso dei pulsanti “OK” o “Annulla”.

### Impostazioni di base

Le impostazioni di base sono una parte molto importante per il corretto funzionamento e uso del sistema. Infatti definendole sarà possibile creare gli esercizi per le specifiche classi che svolgono i moduli e seguono delle precise tematiche.

Questa schermata è composta da tre menu in cui si può scorrere per vedere i dati inseriti (moduli, tematiche e classi) e i relativi pulsanti per aggiungere un nuovo elemento.

L’interfaccia sembra molto semplice, ma dietro ci sono anche alcune funzionalità che permettono di collegare queste tre entità tra loro e gestirle:

* Doppio click su una classe  
  Mostra i moduli che essa svolge e permette di gestirli.
* Doppio click su un modulo  
  Mostra le tematiche che esso comprende e permette di gestirli.
* Click + delete su un elemento lo elimina.

Infine ci sono i soliti pulsanti che consentono di annullare o applicare le eventuali modifiche apportate.

### Esercizi

L’interfaccia che gestisce gli esercizi permette innanzitutto di scegliere tra le informazioni di base, per poi definire i dettagli dell’esercizio, cioè il testo e un eventuale immagine. Gli esercizi salvati verranno memorizzati per poi essere usati durante la realizzazione di una prova.



La prima maschera consente di attribuire un titolo all’esercizio e di scegliere tramite un menu a tendina le informazioni di base stabilite nell’applicazione.

Nell’altra maschera è possibile comporre il testo dell’esercizio e inserire un’immagine attraverso Drag & Drop oppure sfogliando tra i file. Se si clicca “OK” il programma genererà un file RFT con il contenuto del testo e dell’immagine.

### Creazione prova

L’opzione finale dell’applicazione è la creazione delle prove. In questa schermata vengono dapprima definiti i campi fondamentali (più o meno come negli esercizi) e infine vengono selezionati gli esercizi da inserire.



Nel primo campo si scrive il titolo della prova, nel secondo si inserisce la data (anche con l’aiuto del calendario), poi si inseriscono classe e modulo, che vengono estrapolati sempre dalle impostazioni di base. Finalmente si aggiungono gli esercizi tramite Drag & Drop o sfogliando tra i propri file.

Per avere un’anteprima della prova esiste un apposito pulsante che mostrerà un’altra finestra con ciò che produrrà il programma se si dovesse generare il file.

## Design procedurale

In questo capitolo viene spiegato il ciclo di vita del prodotto e il suo comportamento in conseguenza di determinate azioni. Ciò è stato schematizzato in un diagramma di flusso dei dati.



In questo caso il docente è l’unico oggetto che effettua delle azioni di inserimento dati. Le informazioni che quest’ultimo inserisce vengono memorizzate nelle informazioni di base o possono servire per creare un esercizio o una prova.

# Implementazione

Nel capitolo di implementazione viene spiegato in dettaglio come il prodotto finale è stato sviluppato.

## Creazione progetto

Per iniziare si crea una Soluzione Vuota che fungerà da contenitore generale, quindi si inseriscono i progetti del sistema .NET Core, cioè una App WPF e una Libreria di Classi. Il prossimo passo è l’implementazione dei pacchetti necessari per l’amministrazione dei dati tramite il package manager NuGet.



Dall’immagine si può vedere che i pacchetti utilizzati sono quelli inerenti a EntityFrameworkCore, infatti questi sono utili per l’accesso ai dati e la gestione del database (in questo caso SQLite).

## Libreria di Classi

La Libreria di Classi (.NET Core) è uno dei progetti della soluzione Visual Studio e contiene tutto ciò che ha a che fare con la gestione dei dati e la definizione del database.

### Models

La cartella Models contiene le entità di base del sistema che rappresentano i dati, quindi per fare ciò si fa riferimento al diagramma del database. Inoltre le classi modello contengono il metodo ToString() che permette di stampare l’oggetto sotto forma di stringa mostrando il contenuto desiderato (di solito il nome). Infine si può dire che ci sono dei metodi costruttori (di cui uno vuoto per far funzionare i repository) usati per la creazione di nuove entità da inserire nel database. Sono anche presenti le proprietà che rappresentano gli ID delle entità relative in modo da gestire le foreign key durante gli inserimenti di dati nel database.

#### BaseEntity

BaseEntity è la classe modello principale ed è la base di ogni altra entità, infatti essa contiene solo l’Id.

public class BaseEntity

{

public int Id { get; set; }

}

#### Classe

L’entità Classe contiene il nome della stessa (ad esempio “I4AA”), l’anno che sta svolgendo (ad esempio “2019/2020”) e i relativi moduli e prove.

public string Nome { get; set; }

public virtual Anno Anno { get; set; }

public int AnnoId { get; set; }

public virtual ICollection<Modulo> Moduli { get; set; }

public virtual ICollection<Prova> Prove { get; set; }

public Classe(string nome, Anno anno)

{

Nome = nome;

AnnoId = anno.Id;

}

#### Modulo

La classe Modulo presenta i campi che descrivono il nome (“Modulo <numero>”), la classe, le tematiche e gli esercizi. Di quest’ultimi ci sono i campi non mappati che servono per ottenere le liste sotto forma di stringa.

public virtual ICollection<Tematica> Tematiche { get; set; }

public virtual ICollection<Esercizio> Esercizi { get; set; }

[NotMapped]

public string TematicheList => string.Join(", ", Tematiche);

[NotMapped]

public string EserciziList => string.Join(", ", Esercizi);

#### Tematica

Il modello della Tematica contiene il nome (ad esempio “MVC”) e il relativo Modulo.

public Tematica(string nome, Modulo modulo)

{

Nome = nome;

ModuloId = modulo.Id;

}

#### Anno

La classe Anno definisce l’annata (ad esempio “2019/2020”) e la lista delle relative classi.

public string Annata { get; set; }

public virtual ICollection<Classe> Classi { get; set; }

public Anno(string annata) => Annata = annata;

#### Esercizio

L’entità Esercizio rappresenta il titolo, il testo (che conterrà anche l’immagine in quanto è tutto salvato in byte) e i relativi modulo e esercizi della prova.

public virtual ICollection<EsercizioProva> EserciziProva { get; set; }

public Esercizio(string titolo, string testo, Modulo modulo)

{

Titolo = titolo;

Testo = testo;

ModuloId = modulo.Id;

}

public override string ToString() => Titolo;

#### EsercizioProva

La classe EsercizioProva è un’estensione di Esercizio ma rappresenta essenzialmente gli esercizi che vengono inseriti nelle prove, quindi si definisce il relativo esercizio e la prova.

public EsercizioProva(Esercizio esercizio, Prova prova)

{

EsercizioId = esercizio.Id;

ProvaId = prova.Id;

}

public override string ToString() => Esercizio.ToString();

#### Prova

Infine c’è il modello della Prova, il quale contiene il titolo, la data, la classe e gli esercizi (EsercizioProva).

public Prova(string titolo, DateTime data, Classe classe)

{

Titolo = titolo;

Data = data;

ClasseId = classe.Id;

}

### Services

La cartella Services contiene le interfacce e le classi che sono utili per interagire con il database.   
Infatti in queste classi sono presenti i metodi che permettono di inserire, modificare o eliminare i dati.

#### IDataRepository

IDataRepository è l’interfaccia di base che implementa i metodi inerenti alle operazioni sul database. Bisogna definire il modello di dati su cui operare (T) e di conseguenza i metodi lavoreranno in base a quello.

public interface IDataRepository<T> where T : BaseEntity

{

T Get(int id);

IQueryable<T> Get();

T Insert(T entity);

void Update(T entity);

void Delete(T entity);

}

#### DbDataRepository

Questa è la classe che implementa l’interfaccia IDataRepository e viene usata principalmente come base per i repository degli altri modelli di dati implementando i metodi relativi al database. Essenzialmente si definisce il contesto del database e il modello di dati e in base a ciò vengono implementati i metodi per ottenere, inserire, modificare e eliminare i dati.

public abstract class DbDataRepository<C, T> : IDataRepository<T> where T : BaseEntity

where C : DbContext, new()

{

protected C context;

protected DbDataRepository(C ctx) => context = ctx;

public T Get(int id) => Get().SingleOrDefault(be => be.Id == id);

public virtual IQueryable<T> Get() => context.Set<T>();

public virtual T Insert(T entity)

{

context.Set<T>().Add(entity);

context.SaveChanges();

return entity;

}

public virtual void Update(T entity)

{

context.Entry(entity).State = EntityState.Modified;

context.SaveChanges();

}

public virtual void Delete(T entity)

{

context.Set<T>().Remove(entity);

context.SaveChanges();

}

}

#### IClasseRepository

Interfaccia figlia di IDataRepository relativa al modello di dati della classe.

public interface IClasseRepository : IDataRepository<Classe> { }

#### ClasseDbRepository

Classe figlia di DbDataRepository relativa al modello di dati della classe. È presente il metodo che ritorna tutte le entità ordinate per l’anno.

public class ClasseDbRepository : DbDataRepository<AppDbContext, Classe>, IClasseRepository

{

public ClasseDbRepository(AppDbContext ctx) : base(ctx) { }

public override IQueryable<Classe> Get() => base.Get().OrderBy(s => s.Anno);

}

#### IModuloRepository

Interfaccia figlia di IDataRepository relativa al modello di dati del modulo.

public interface IModuloRepository : IDataRepository<Modulo> { }

#### ModuloDbRepository

Classe figlia di DbDataRepository relativa al modello di dati della classe. Viene implementato il metodo che ritorna tutti i moduli ordinati per nome.

public class ModuloDbRepository : DbDataRepository<AppDbContext, Modulo>, IModuloRepository

{

public ModuloDbRepository(AppDbContext ctx) : base(ctx) { }

public override IQueryable<Modulo> Get() => base.Get().OrderBy(s => s.Nome);

}

#### ITematicaRepository

Interfaccia figlia di IDataRepository relativa al modello di dati della tematica.

public interface ITematicaRepository : IDataRepository<Tematica> { }

#### TematicaDbRepository

Classe figlia di DbDataRepository relativa al modello di dati della tematica. Viene ereditato il metodo Get() in modo da ottenere tutte le tematiche ordinate per il nome del relativo modulo.

public class TematicaDbRepository : DbDataRepository<AppDbContext, Tematica>, ITematicaRepository

{

public TematicaDbRepository(AppDbContext ctx) : base(ctx) { }

public override IQueryable<Tematica> Get() => base.Get().OrderBy(s => s.Modulo.Nome);

}

#### IAnnoRepository

Interfaccia figlia di IDataRepository relativa al modello di dati dell’anno.

public interface IAnnoRepository : IDataRepository<Anno> { }

#### AnnoDbRepository

Classe figlia di DbDataRepository per il modello di dati dell’anno. C’è il metodo che ritorna tutte le entità ordinate per l’annata.

public class AnnoDbRepository : DbDataRepository<AppDbContext, Anno>, IAnnoRepository

{

public AnnoDbRepository(AppDbContext ctx) : base(ctx) { }

public override IQueryable<Anno> Get() => base.Get().OrderByDescending(s => s.Annata);

}

#### IEsercizioRepository

Interfaccia figlia di IDataRepository relativa al modello di dati dell’esercizio.

public interface IEsercizioRepository : IDataRepository<Esercizio> { }

#### EsercizioDbRepository

Classe figlia di DbDataRepository relativa al modello di dati dell’esercizio. È presente il metodo che ritorna gli esercizi ordinati per titolo.

public class EsercizioDbRepository : DbDataRepository<AppDbContext, Esercizio>, IEsercizioRepository

{

public EsercizioDbRepository(AppDbContext ctx) : base(ctx) { }

public override IQueryable<Esercizio> Get() => base.Get().OrderBy(s => s.Titolo);

}

#### IEsercizioProvaDbRepository

Interfaccia figlia di IDataRepository che implementa i metodi relativi al modello degli esercizi della prova.

public interface IEsercizioProvaRepository : IDataRepository<EsercizioProva> { }

#### EsercizioProvaDbRepository

DbDataRepository relativo al modello di dati dell’esercizio della prova.

public class EsercizioProvaDbRepository : DbDataRepository<AppDbContext, EsercizioProva>, IEsercizioProvaRepository

{

public EsercizioProvaDbRepository(AppDbContext ctx) : base(ctx) { }

}

#### IProvaRepository

Interfaccia figlia di IDataRepository relativa al modello di dati della prova.

public interface IProvaRepository : IDataRepository<Prova> { }

#### ProvaDbRepository

Classe figlia di DbDataRepository relativa al modello di dati della prova.

public class ProvaDbRepository : DbDataRepository<AppDbContext, Prova>, IProvaRepository

{

public ProvaDbRepository(AppDbContext ctx) : base(ctx) { }

public override IQueryable<Prova> Get() => base.Get().OrderBy(s => s.Data);

}

### AppDbContext

AppDbContext è la classe che praticamente mette insieme i modelli di dati (Models) e le interfacce con il database (Services) per impostare la base di dati configurando le raccolte delle entità e il percorso di memorizzazione SQLite.

public class AppDbContext : DbContext

{

public DbSet<Classe> Classi { get; set; }

public DbSet<Modulo> Moduli { get; set; }

public DbSet<Tematica> Tematiche { get; set; }

public DbSet<Anno> Anni { get; set; }

public DbSet<Esercizio> Esercizi { get; set; }

public DbSet<EsercizioProva> EserciziProva { get; set; }

public DbSet<Prova> Prove { get; set; }

public AppDbContext() : base() { }

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

string dbPath =

Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData)

+ "\\GestioneEsercizi";

Directory.CreateDirectory(dbPath);

optionsBuilder.UseLazyLoadingProxies().UseSqlite(

"Data Source=" + dbPath + "\\GestioneEsercizi.sqlite");

}

}

}

Le proprietà della classe (DbSet) rappresentano le tabelle della base di dati, mentre il metodo OnConfiguring imposta il percorso di memorizzazione prendendo come cartella di base quella di AppData. Successivamente è necessario creare il database tramite la console di gestione dei pacchetti. La prima cosa da fare è la creazione delle migrazioni, le quali definiscono i parametri del database tramite il DbContext, per poi aggiornare la base di dati. I comandi da eseguire sono i seguenti (Microsoft.EntityFrameworkCore.Tools):

* Add-Migration Initial
* Update-Database

#### Database

Il database definitivo ha subito qualche modifica rispetto a quello inizialmente progettato.



Ciò che è cambiato è il fatto che essenzialmente le entità della classe e quella dell’anno sono state scambiate tra di loro, rendendo più chiara la struttura visto che i collegamenti che l’anno creava non erano ben definiti.

## App WPF

L’App WPF (.NET Core) è il progetto della soluzione Visual Studio che gestisce i dati della libreria di classi mostrandoli su un’interfaccia grafica user-friendly e permettendo di operare appunto sui dati.

### ViewModels

La cartella ViewModels contiene le classi che collegano i modelli di dati della libreria di classi con il progetto WPF permettendo di mostrarli nell’interfaccia grafica. Per fare ciò si usa la classe base BindableBase, la quale presenta i metodi principali per l’implementazione di un ViewModel.

/// <summary>

/// Base class for all ViewModel classes in the application. Provides support for

/// property changes notification. Original implementation by Josh Smith.

/// </summary>

public abstract class BindableBase : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged = delegate { };

protected void OnPropertyChanged(string propertyName)

{

PropertyChangedEventHandler handler = PropertyChanged;

if (handler != null)

{

handler(this, new PropertyChangedEventArgs(propertyName));

}

}

protected virtual void SetProperty<T>

(ref T member, T value, [CallerMemberName] string propertyName = null)

{

if (object.Equals(member, value))

return;

member = value;

OnPropertyChanged(propertyName);

}

protected void OnPropertyChanged<T>(Expression<Func<T>> propertyExpression)

{

var body = propertyExpression.Body as MemberExpression;

var expression = body.Expression as ConstantExpression;

PropertyChanged(expression.Value, new PropertyChangedEventArgs(body.Member.Name));

}

}

#### MainViewModel

Il ViewModel principale ha il compito di impostare gli altri ViewModels e gestirli tramite i Commands e la proprietà CurrentViewModel. Qui è documentata la soluzione simbolica, visto che in realtà nel progetto ci sono molti ViewModels ma comunque il sistema funziona istanziando tutti i ViewModels e i Commands relativi alla schermata corrente (ad esempio se una schermata può portare ad altre due schermate allora conterrà due Commands).

public class MainViewModel : BindableBase

{

private ViewModel vm;

private AltroViewModel avm;

public IDelegateCommand AltroCommand { get; set; }

private BindableBase currentViewModel;

public BindableBase CurrentViewModel

{

get { return currentViewModel; }

set { SetProperty(ref currentViewModel, value); }

}

public MainViewModel()

{

ViewModel = new ViewModel();

AltroViewModel = new AltroViewModel();

CurrentViewModel = ViewModel;

AltroCommand = new DelegateCommand(OnAltro, CanAltro);

// Ricezione dei messaggi da parte degli altri ViewModels.

Messenger.Default.Register<BindableBase>(this, OnViewModelReceived);

}

public void OnViewModelReceived(BindableBase viewmodel) => CurrentViewModel = viewmodel;

private void OnAltro(object obj) => CurrentViewModel = AltroViewModel;

private bool CanAltro(object arg) => true;

}

In questo caso viene utilizzata la classe Messenger che permette l’invio e la ricezione di dati tra ViewModels. Viene fatto ciò perché è necessario basare tutto il sistema sul MainViewModel e la proprietà CurrentViewModel, in quanto le altre classi mandano il messaggio contenente il ViewModel da utilizzare.

#### BenvenutoViewModel

ViewModel relativo alla schermata di benvenuto, si impostano i ViewModels delle funzioni principali del programma (configurazione impostazioni di base, nuovo esercizio, creazione prova).

public BenvenutoViewModel()

{

ImpostazioniBaseCommand = new DelegateCommand(OnImpostazioniBase, CanImpostazioniBase);

EsercizioCommand = new DelegateCommand(OnEsercizio, CanEsercizio);

ProvaCommand = new DelegateCommand(OnProva, CanProva);

}

Inoltre viene usato il Messenger per selezionare il ViewModel da far mostrare al MainViewModel.

private void OnProva(object obj) => Messenger.Default.Send<BindableBase>(new ViewModel());

#### AboutViewModel

ViewModel relativo alla schermata di informazioni sul prodotto, si imposta il ViewModel della schermata di benvenuto.

public IDelegateCommand BenvenutoCommand { get; set; }

public AboutViewModel()

{

BenvenutoCommand = new DelegateCommand(OnBenvenuto, CanBenvenuto);

}

private void OnBenvenuto(object obj)

{

Messenger.Default.Send<BindableBase>(new BenvenutoViewModel());

}

#### GuidaViewModel

Rappresenta la schermata con la guida del prodotto, viene solo impostato il primo ViewModel.

public GuidaViewModel()

{

BenvenutoCommand = new DelegateCommand(OnBenvenuto, CanBenvenuto);

}

#### EsercizioListViewModel

ViewModel utile per mostrare la lista degli esercizi memorizzati.

public IDelegateCommand BenvenutoCommand { get; set; }

public ObservableCollection<Esercizio> Esercizi { get; set; }

public EsercizioListViewModel()

{

BenvenutoCommand = new DelegateCommand(OnBenvenuto, CanBenvenuto);

EsercizioDbRepository repo = new EsercizioDbRepository(new AppDbContext());

Esercizi = new ObservableCollection<Esercizio>(repo.Get());

}

#### ProvaListViewModel

Simile a EsercizioListViewModel però vengono mostrate le prove memorizzate anziché gli esercizi.

public ProvaListViewModel()

{

ProvaDbRepository repo = new ProvaDbRepository(new AppDbContext());

Prove = new ObservableCollection<Prova>(repo.Get());

}

#### ImpostazioniBaseViewModel

ImpostazioniBaseViewModel ha il principale scopo di impostare le liste delle tre impostazioni di base. Inoltre sono presenti tre pulsanti che portano ai rispettivi ViewModels dell’oggetto che si vuole inserire.

public ImpostazioniBaseViewModel()

{

BenvenutoCommand = new DelegateCommand(OnBenvenuto, CanBenvenuto);

ClasseCommand = new DelegateCommand(OnClasse, CanClasse);

ModuloCommand = new DelegateCommand(OnModulo, CanModulo);

TematicaCommand = new DelegateCommand(OnTematica, CanTematica);

AppDbContext ctx = new AppDbContext();

ClasseDbRepository repoClasse = new ClasseDbRepository(ctx);

ModuloDbRepository repoModulo = new ModuloDbRepository(ctx);

TematicaDbRepository repoTematica = new TematicaDbRepository(ctx);

Classi = new ObservableCollection<Classe>(repoClasse.Get());

Moduli = new ObservableCollection<Modulo>(repoModulo.Get());

Tematiche = new ObservableCollection<Tematica>(repoTematica.Get());

}

#### ClasseViewModel

Questo ViewModel è utile per implementare l’inserimento di una nuova classe.

public ClasseViewModel()

{

BenvenutoCommand = new DelegateCommand(OnBenvenuto, CanBenvenuto);

SalvaCommand = new DelegateCommand(OnSalva, CanSalva);

AnnoDbRepository repo = new AnnoDbRepository(new AppDbContext());

Anni = new ObservableCollection<Anno>(repo.Get());

}

private void OnSalva(object obj)

{

ClasseDbRepository repo = new ClasseDbRepository(new AppDbContext());

// Aggiungo la classe

repo.Insert(new Classe(Nome, Anno));

OnBenvenuto(obj);

}

#### ModuloViewModel

Funziona come ClasseViewModel ma per l’inserimento di un nuovo modulo.

public ModuloViewModel()

{

BenvenutoCommand = new DelegateCommand(OnBenvenuto, CanBenvenuto);

SalvaCommand = new DelegateCommand(OnSalva, CanSalva);

ClasseDbRepository repo = new ClasseDbRepository(new AppDbContext());

Classi = new ObservableCollection<Classe>(repo.Get());

}

private void OnSalva(object obj)

{

ModuloDbRepository repo = new ModuloDbRepository(new AppDbContext());

// Aggiungo il modulo

repo.Insert(new Modulo(Nome, Classe));

OnBenvenuto(obj);

}

#### TematicaViewModel

ViewModel simile a ClasseViewModel ma viene inserita una tematica anziché una classe.

private void OnSalva(object obj)

{

TematicaDbRepository repo = new TematicaDbRepository(new AppDbContext());

// Aggiungo la tematica

repo.Insert(new Tematica(Nome, Modulo));

OnBenvenuto(obj);

}

#### EsercizioViewModel

Questo è il ViewModel che gestisce la creazione di un nuovo esercizio. In questo caso vengono gestiti i due comandi per l’annullamento e il salvataggio dell’operazione, viene creata la lista di moduli e vengono reperiti dalla View i dati utili per creare un nuovo esercizio (titolo, testo e modulo).

public string Titolo { get; set; }

public string Testo { get; set; }

public Modulo Modulo { get; set; }

public EsercizioViewModel()

{

BenvenutoCommand = new DelegateCommand(OnBenvenuto, CanBenvenuto);

SalvaCommand = new DelegateCommand(OnSalva, CanSalva);

ModuloDbRepository repoModulo = new ModuloDbRepository(new AppDbContext());

Moduli = new ObservableCollection<Modulo>(repoModulo.Get());

}

Infine viene implementato il salvataggio dell’esercizio nel database.

private void OnSalva(object obj)

{

EsercizioDbRepository repoEsercizio = new EsercizioDbRepository(new AppDbContext());

// Aggiungo l'esercizio

repoEsercizio.Insert(new Esercizio(Titolo, Testo, Modulo));

OnBenvenuto(obj);

}

#### ProvaViewModel

ViewModel che gestisce la creazione di una prova. In generale questa classe è simile al ViewModel dell’esercizio in quanto le operazioni da fare sono più o meno le stesse, però in questo caso bisogna anche gestire il salvataggio degli esercizi della prova (EsercizioProva).

public string Titolo { get; set; }

public DateTime Data { get; set; }

public Anno Anno { get; set; }

public ProvaViewModel()

{

BenvenutoCommand = new DelegateCommand(OnBenvenuto, CanBenvenuto);

SalvaCommand = new DelegateCommand(OnSalva, CanSalva);

AnnoDbRepository repoAnno = new AnnoDbRepository(new AppDbContext());

EsercizioDbRepository repoEsercizio = new EsercizioDbRepository(new AppDbContext());

Anni = new ObservableCollection<Anno>(repoAnno.Get());

Esercizi = new ObservableCollection<Esercizio>(repoEsercizio.Get());

}

Dopodiché c’è il metodo che implementa il salvataggio della prova.

private void OnSalva(object obj)

{

ProvaDbRepository repoProva = new ProvaDbRepository(new AppDbContext());

EsercizioProvaDbRepository repo = new EsercizioProvaDbRepository(new AppDbContext());

// Aggiungo la prova

Prova prova = new Prova(Titolo, Data, Anno);

repoProva.Insert(prova);

// Aggiungo gli esercizi della prova

foreach (Esercizio esercizio in new List<Esercizio>(Esercizi))

{

repo.Insert(new EsercizioProva(esercizio, prova));

}

OnBenvenuto(obj);

}

### Views

La cartella Views rappresenta gli elementi dell’interfaccia utente dell’applicazione. La struttura di base funziona in modo che ogni View utilizzi il relativo ViewModel come base per i dati passandosi le informazioni tramite il Messenger.

#### MainView

La MainView è l’interfaccia principale e ha il compito di rappresentare il MainViewModel, quindi viene selezionata la View da mostrare attraverso i Commands e il campo CurrentViewModel.

<UserControl.DataContext>

<viewmodel:MainViewModel/>

</UserControl.DataContext>

<Menu Grid.Row="0">

<MenuItem Header="\_File">

<MenuItem Header="\_Esercizi" Command="{Binding Path=EsercizioListCommand}"/>

<MenuItem Header="\_Prove" Command="{Binding Path=ProvaListCommand}"/>

<MenuItem x:Name="miEsci" Header="\_Esci" Click="miEsci\_Click"/>

</MenuItem>

<MenuItem Header="\_Info">

<MenuItem Header="\_Guida" Command="{Binding Path=GuidaCommand}"/>

<MenuItem Header="\_About" Command="{Binding Path=AboutCommand}"/>

</MenuItem>

</Menu>

<ContentControl Grid.Row="1" Content="{Binding Path=CurrentViewModel}"/>

Sono inoltre presenti i riferimenti di tutti gli altri ViewModels e le rispettive Views nella sezione delle risorse.

<UserControl.Resources>

<DataTemplate DataType="{x:Type viewmodel:BenvenutoViewModel}">

<local:BenvenutoView/>

</DataTemplate>

Questa View sarà l’unico contenuto presente nella finestra principale del progetto, cioè MainWindow.

<Window x:Class="GestioneEsercizi.MainWindow"

Title="Gestione Esercizi" Height="500

<Grid>

<view:MainView/>

</Grid>

</Window>

#### BenvenutoView

View che rappresenta la schermata principale di benvenuto.

<Button Grid.Column="1"

VerticalAlignment="Center" HorizontalAlignment="Center"

Content="Nuovo Esercizio" Padding="10"

Command="{Binding Path=EsercizioCommand}"/>

Sono presenti i tre pulsanti che permettono di aprire le schermate relative alle operazioni principali, quindi la gestione delle impostazioni di base, la creazione di un nuovo esercizio e la creazione di una nuova prova.

#### AboutView

View relativa alla schermata delle informazioni del prodotto.

<StackPanel VerticalAlignment="Center" HorizontalAlignment="Center">

<Image Source="/Images/GA.png" Width="100" Margin="20"/>

<TextBlock HorizontalAlignment="Center">

<Hyperlink RequestNavigate="NavigateWebsite"

NavigateUri="https://github.com/gabrialessi/GestioneEsercizi">

GitHub - GestioneEsercizi

</Hyperlink>

</TextBlock>

In questo caso viene mostrata un’immagine, il link del repository del progetto su GitHub, il link alla pagina web dell’autore e la versione.

#### GuidaView

View che rappresenta la schermata della guida del prodotto.

<Label FontSize="25" FontWeight="Bold" Content="Guida - Gestione Esercizi"/>

<Label FontSize="20" Content="Introduzione"/>

<Label FontSize="20" Content="Impostazioni di Base"/>

<Label FontSize="20" Content="Nuovo Esercizio"/>

<Label FontSize="20" Content="Creazione Prova"/>

In questo caso viene solo impostato del testo a scopo informativo.

#### EsercizioListView

Questa View ha il semplice scopo di mostrare la lista di esercizi presenti nel database.

<DataGrid x:Name="eserciziDataGrid" Grid.Row="0" IsReadOnly="True"

ItemsSource="{Binding Path=Esercizi}" AutoGenerateColumns="False">

<DataGrid.Columns>

<DataGridTextColumn Header="Titolo" Binding="{Binding Path=Titolo}" />

<DataGridTextColumn Header="Modulo" Binding="{Binding Path=Modulo}" />

</DataGrid.Columns>

</DataGrid>

#### ProvaListView

ProvaListView funziona come EsercizioListView però mostra le prove anziché gli esercizi.

<DataGrid x:Name="proveDataGrid" Grid.Row="0" IsReadOnly="True"

ItemsSource="{Binding Path=Prove}" AutoGenerateColumns="False">

<DataGrid.Columns>

<DataGridTextColumn Header="Titolo" Binding="{Binding Path=Titolo}" />

<DataGridTextColumn Header="Data" Binding="{Binding Path=Data}" />

<DataGridTextColumn Header="Anno" Binding="{Binding Path=Anno}" />

</DataGrid.Columns>

</DataGrid>

#### ImpostazioniBaseView

View che mostra le impostazioni di base presenti nel sistema e permette di gestirle inserendo nuove entità. Le tabelle vengono istanziate in base ai singoli campi presenti nel ViewModel, così facendo le informazioni vengono mostrate correttamente.

<DataGrid IsReadOnly="True" AutoGenerateColumns="False"

ItemsSource="{Binding Path=Classi}">

<DataGrid.Columns>

<DataGridTextColumn Header="Classe" Binding="{Binding Path=Nome}" />

<DataGridTextColumn Header="Anno" Binding="{Binding Path=Anno}" />

</DataGrid.Columns>

</DataGrid>

Quando si vuole aggiungere un nuovo elemento viene chiamata la rispettiva View che permette di effettuare l’inserimento basandosi sul sistema dei dati.

<Button Grid.Column="0" Grid.Row="2" Content="Nuova classe"

Command="{Binding Path=ClasseCommand}"/>



#### EsercizioView

Gli elementi presenti in questa View permettono di creare un nuovo esercizio. Vengono definite le impostazioni di base, il titolo, il testo e un’eventuale immagine. La seguente parte di XML è utile per creare del testo di suggerimento che viene mostrato quando il campo è vuoto e viene usato per il titolo e per il testo.

<!-- Titolo esercizio -->

<TextBox x:Name="titoloTextBox" Grid.Column="0" Grid.Row="0" />

<!-- Placeholder -->

<TextBlock IsHitTestVisible="False" Text="Titolo esercizio..."

VerticalAlignment="Center" Foreground="DarkGray" Margin="10 0 0 0">

<TextBlock.Style>

<Style TargetType="{x:Type TextBlock}">

<Setter Property="Visibility" Value="Collapsed"/>

<Style.Triggers>

<DataTrigger Binding="{Binding Text, ElementName=titoloTextBox}" Value="">

<Setter Property="Visibility" Value="Visible"/>

</DataTrigger>

</Style.Triggers>

</Style>

</TextBlock.Style>

</TextBlock>

Dopodiché ci sono i menu a tendina che permettono di selezionare le impostazioni di base.

<!-- Menu - Modulo, Tematica, Classe -->

<ComboBox Grid.Column="1" Grid.Row="0" SelectedIndex="0"

ItemsSource="{Binding Path=Moduli}" />

<ComboBox Grid.Column="2" Grid.Row="0" SelectedIndex="0"

ItemsSource="{Binding Path=Tematiche}" />

<ComboBox Grid.Column="3" Grid.Row="0" SelectedIndex="0"

ItemsSource="{Binding Path=Classi}" />

Infine c’è un contenitore che consente di trascinare l’immagine oppure selezionarla manualmente tramite il file explorer.

<!-- Immagine -->

<Border Grid.Column="2" Grid.ColumnSpan="2" Grid.Row="1"

BorderBrush="Black" BorderThickness="0.5" Margin="50"

AllowDrop="True" Drop="ImageDrop">

<Grid>

<Label x:Name="immagineLabel" Content="Trascina immagine" />

<Button x:Name="sfogliaButton" VerticalAlignment="Center"

HorizontalAlignment="Center" Content="Sfoglia..." Click="sfogliaButtonClick" />

</Grid>

</Border>

Poi ovviamente sono presenti i pulsanti “Annulla” e “Salva”.

<DockPanel Grid.Row="2" Grid.Column="3" Margin="10"

VerticalAlignment="Bottom" HorizontalAlignment="Right">

<!-- Pulsante salva -->

<Button Content="Annulla" Command="{Binding Path=BenvenutoCommand}" />

<!-- Pulsante indietro -->

<Button Content="Salva" Margin="10 0 0 0" />

</DockPanel>

Nella parte di C# della View vengono implementati i metodi relativi all’inserimento e al trascinamento dell’immagine. Nel primo caso viene usato un oggetto che rappresenta il file explorer al quale viene impostato un filtro in modo da mostrare solo le immagini.

private void sfogliaButtonClick(object sender, RoutedEventArgs e)

{

Microsoft.Win32.OpenFileDialog dlg = new Microsoft.Win32.OpenFileDialog();

dlg.Filter = "Image Files | \*.jpg; \*.jpeg; \*.png; \*.gif";

if (dlg.ShowDialog() == true) SetLabelContent(dlg.FileName);

}

Invece per quanto riguarda il rilascio di un’immagine vengono utilizzato i parametri relativo metodo in modo da ottenere l’ultimo file trascinato.

private void ImageDrop(object sender, DragEventArgs e)

{

if (e.Data.GetDataPresent(DataFormats.FileDrop))

{

string[] files = (string[])e.Data.GetData(DataFormats.FileDrop);

SetLabelContent(files[0]);

}

}



#### ProvaView

La ProvaView è simile a EsercizioView visto che lo scopo è un più o meno lo stesso, cioè creare un nuovo elemento in base ai parametri presenti nel database. Anche in questo caso c’è la parte di inserimento del titolo dove è presente l’implementazione del placeholder.

<TextBox x:Name="titoloTextBox" Grid.Column="0" Grid.Row="0"

VerticalAlignment="Center" Text="{Binding Path=Titolo}"/>

Poi c’è un campo dove si può selezionare la data esatta della prova.

<DatePicker x:Name="dataDatePicker" Grid.Column="0" Grid.Row="1"

VerticalAlignment="Center" SelectedDate="{Binding Path=Data}" />

Per definire l’annata della nuova prova si utilizzano le entità presenti nel database creando un menu a tendina.

<ComboBox Grid.Column="0" Grid.Row="2" SelectedIndex="0" VerticalAlignment="Center"

ItemsSource="{Binding Path=Anni}" SelectedItem="{Binding Path=Anno}" />

Infine per scegliere gli esercizi da inserire viene usata una tabella in cui si selezionano quelli da inserire.

<DataGrid x:Name="eserciziDataGrid" Grid.Column="1" Grid.Row="0" Grid.RowSpan="3" Margin="5"

IsReadOnly="True" ItemsSource="{Binding Path=Esercizi}" AutoGenerateColumns="False">

<DataGrid.Columns>

<DataGridTextColumn Header="Titolo" Binding="{Binding Path=Titolo}" />

<DataGridTextColumn Header="Modulo" Binding="{Binding Path=Modulo}" />

<DataGridCheckBoxColumn Header="Inserire nella prova" IsReadOnly="False" />

</DataGrid.Columns>

</DataGrid>

# Test

## Protocollo di test

Definire in modo accurato tutti i test che devono essere realizzati per garantire l’adempimento delle richieste formulate nei requisiti. I test fungono da garanzia di qualità del prodotto. Ogni test deve essere ripetibile alle stesse condizioni.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case:**  **Riferimento**: | TC-001  REQ-012 | **Nome:** | Import a card with KIC, KID and KIK keys, but not shown with the GUI |
| **Descrizione:** | Import a card with KIC, KID and KIK keys with no obfuscation, but not shown with the GUI | | |
| **Prerequisiti:** | Store on local PC: Profile\_1.2.001.xml (appendix n\_n) and Cards\_1.2.001.txt (appendix n\_n).  PIN (OTA\_VIEW\_PIN\_PUK\_KEY) and ADM (OTA\_VIEW\_ADM\_KEY) user right not set. | | |
| **Procedura:** | 1. Go to “Cards manager” menu,  in main page click “Import Profiles” link, Select the “1.2.001.xml” file, Import the Profile 2. Go to “Cards manager” menu,  in main page click “Import Cards” link, Select the “1.2.001.txt” file, Delete the cards,  Select the “1.2.001.txt” file, Import the cards 3. Research the “41795924770” Card, Click the imsi card link Check the card details 4. Execute the SQL: SELECT imsi, dir, keyset, cntr, rawtohex(kickey), rawtohex(kidkey), rawtohex(kikkey), rawtohex(chv), rawtohex(dap)FROM otacardkey a where imsi='340041795924770' ORDER BY keyset; | | |
| **Risultati attesi:** | Keys visible in the DB (OtaCardKey) but not visible in the GUI (Card details) | | |

## Risultati test

Tabella riassuntiva in cui si inseriscono i test riusciti e non del prodotto finale. Se un test non riesce e viene corretto l’errore, questo dovrà risultare nel documento finale come riuscito (la procedura della correzione apparirà nel diario), altrimenti dovrà essere descritto l’errore con eventuali ipotesi di correzione.

## Mancanze/limitazioni conosciute

Descrizione con motivazione di eventuali elementi mancanti o non completamente implementati, al di fuori dei test case. Non devono essere riportati gli errori e i problemi riscontrati e poi risolti durante il progetto.

# Consuntivo

Consuntivo del tempo di lavoro effettivo e considerazioni riguardo le differenze rispetto alla pianificazione (cap 1.7) (ad esempio Gannt consuntivo).

# Conclusioni

Quali sono le implicazioni della mia soluzione? Che impatto avrà? Cambierà il mondo? È un successo importante? È solo un’aggiunta marginale o è semplicemente servita per scoprire che questo percorso è stato una perdita di tempo? I risultati ottenuti sono generali, facilmente generalizzabili o sono specifici di un caso particolare? ecc

## Sviluppi futuri

Migliorie o estensioni che possono essere sviluppate sul prodotto.

## Considerazioni personali

Cosa ho imparato in questo progetto? ecc

# Bibliografia

## Sitografia

* <https://fmoralesdev.com/2019/05/16/generate-class-diagram-vs2019-net-core/>, Generate a class diagram in VS2019, 10.09.2019
* <https://www.c-sharpcorner.com/article/difference-between-net-framework-and-net-core/>, Difference Between .NET Framework and .NET Core, 27.09.2019
* <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/documentation-comments>, Documentation comments, 03.10.2019
* <https://stackoverflow.com/questions/9936796/create-a-menu-bar-in-wpf>, Create a menu Bar in WPF, 15.10.2019
* <https://stackoverflow.com/questions/2820357/how-do-i-exit-a-wpf-application-programmatically>,   
  Exit a WPF application programmatically, 15.10.2019
* <https://stackoverflow.com/questions/11133947/how-do-i-open-a-second-window-from-the-first-window-in-wpf>, Open a second window from the first window in WPF, 15.10.2019
* <https://stackoverflow.com/questions/3419909/how-do-i-lock-a-wpf-window-so-it-can-not-be-moved-resized-minimized-maximized?rq=1>, Lock WPF Window, 15.10.2019
* <https://stackoverflow.com/questions/24485197/relative-path-not-working-while-accessing-a-sqlite-database-through-c-sharp>, Relative Path, 25.10.2019
* <https://stackoverflow.com/questions/833943/watermark-hint-text-placeholder-textbox>, Hint text, 07.11.2019
* <https://stackoverflow.com/questions/10315188/open-file-dialog-and-select-a-file-using-wpf-controls-and-c-sharp>, Open file dialog and select a file using WPF controls and C#, 07.11.2019
* <https://stackoverflow.com/questions/50180326/how-to-make-lazy-loading-work-with-ef-core-2-1-0-and-proxies/52432651>, Lazy Loading with Proxies EF Core, 12.11.2019
* <https://docs.microsoft.com/en-us/ef/core/modeling/included-properties>, Including & Excluding Properties, 19.11.2019
* <https://stackoverflow.com/questions/14959824/convert-list-into-comma-separated-string/29575082>, Convert List into Comma-Separated String, 19.11.2019
* <https://www.wpf-tutorial.com/datagrid-control/custom-columns/>, DataGrid columns, 21.11.2019

# Allegati

Elenco degli allegati, esempio:

* Diari di lavoro
* Codici sorgente/documentazione macchine virtuali
* Istruzioni di installazione del prodotto (con credenziali di accesso) e/o di eventuali prodotti terzi
* Documentazione di prodotti di terzi
* Eventuali guide utente / Manuali di utilizzo
* Mandato e/o Qdc
* Prodotto
* …