

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Sistemi Informativi T

***SymbolEditor: Creazione dinamica di
un plugin per CAD generico***

CANDIDATO

Angelo Maximilian Tulbure

RELATORE

Prof. Paolo Ciaccia

CORRELATORE

Ing. Mattia D'Arpa

Anno Accademico 2021/2022

Sessione II

a me,

alla mia famiglia

e a chi mi è stato vicino in questo percorso

Indice

Introduzione	6
MeXage	8
1 Meta Programmazione e CAD	9
1.1 Definizione Meta Programmazione	9
1.2 Diversi tipi di approccio alla Meta Programmazione	11
1.3 CAD	12
1.3.1 CADdy	12
2 Pianificazione e Fasi Iniziali	14
2.1 Pianificazione del progetto	14
2.2 Suddivisione in sottoprogetti	15
2.3 Fasi iniziali del progetto	15
3 Template XML, NuGet Packages e CommandLineParser	18
3.1 Template XML per l'estrazione dei dati	18
3.2 Import Package Source e NuGet Packages	21
3.3 Inserimento dati da parte dell'utente	24
4 Deserializzazione e Serializzazione del file XML	25
4.1 Deserializzazione del file XML	25
4.2 Serializzazione del file XML	26
4.3 Visitor Design Pattern	29
5 Creazione zip GA e log	33
5.1 Serializzazione JSON files e creazione ZIP GA	33
5.2 Log4net	36
6 GUI e Testing	38
6.1 Introduzione GUI SymbolEditor	38
6.2 WPF e XAML	38
6.3 Sezioni GUI	39
6.3.1 Sezione General	42
6.3.2 Sezione Input	44

6.3.3 Sezione Type	45
6.3.4 Sezione View	46
6.4 Testing	48
7 Conclusioni e Sviluppi Futuri	49
Bibliografia e Sitografia	50
Indice Figure	51
Ringraziamenti	53

Introduzione

Grazie all'uso dei software CAD di progettazione 2D e 3D ingegneri, architetti, topografi e molte altre figure professionali hanno velocizzato e semplificato lo sviluppo, la modifica e l'ottimizzazione del processo di progettazione. I CAD hanno consentito di eseguire rappresentazioni più accurate e modificarle facilmente per migliorare la qualità del progetto. Oggi, i progetti possono essere archiviati sul cloud, dunque interi team possono verificare in tempo reale lo sviluppo del progetto. L'utilizzo efficace di tutte queste features in definitiva ha aumentato la produttività.

L'obiettivo di questo elaborato è quello di analizzare lo sviluppo, sia della parte di back end che di front end, di un plugin per CAD 2D generico che consenta la creazione e l'inserimento di nuovi oggetti dentro il CAD stesso.

Nello specifico è stato sviluppato un plugin che, prendendo un file XML formattato in modo specifico, ne deserializzi il contenuto, lo riserializzi e ne crei un DLL pronto all'uso nel CAD stesso.

Lo scopo finale del progetto è creare automaticamente una cartella con path specificato dall'utente al cui interno è presente un file compresso con estensione GA. Questo file contiene al suo interno il DLL, una sottocartella contenente gli SVG ed i file "map.json" e "manifest.json". Grazie a questo file GA il software "CADdy" riesce a creare dei nuovi oggetti con tutte le proprietà specificate dall'utente nel file XML e le rende disponibili e usabili dall'utente nel software stesso. CADdy è un software sviluppato da MeXage, azienda in cui è stato portato avanti il progetto di tesi.

È stato quindi necessario un approccio alla meta programmazione, per non aver bisogno della presenza umana nella conversione del file XML in oggetto pronto all'uso in CADdy. In realtà questo progetto si può definire "circa" meta programmazione, è stato sviluppato un editor grafico per la scrittura di codice. Al posto di un IDE classico in cui scrivere del testo che verrà riconosciuto come codice, è stato ideato un editor grafico che permette di definire degli oggetti che poi verranno tradotti in codice.

Per una più facile comprensione del progetto è stato ideato uno schema riassuntivo che comprende tutte le fasi più importanti, dalla fase iniziale, ovvero la scelta della struttura del file XML, fino alla fase finale, ovvero la creazione di nuovi componenti da usare nel CAD. Le fasi intermedie sono rappresentate dalla deserializzazione del file XML, dalla sua conseguente

riserializzazione in classe, la compilazione del progetto ed il file compresso con estensione GA contenente i file citati in precedenza.

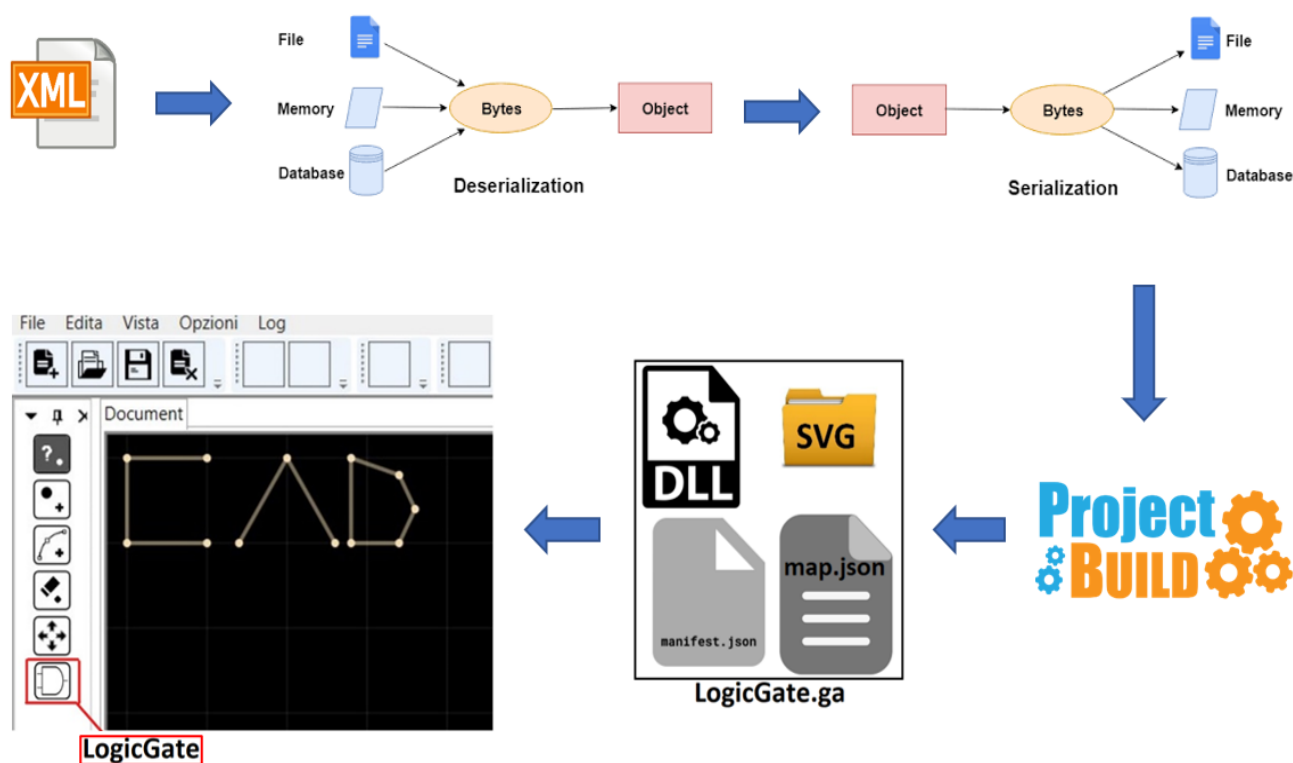


Figura 1 – Schema semplificato progetto

Nei seguenti capitoli di questo elaborato analizzeremo in dettaglio ogni singolo passaggio e ci soffermeremo per approfondire le parti più interessanti ed ostiche.

Nella parte finale dell'elaborato, invece, sarà mostrata con un esempio pratico l'interfaccia grafica che permetterà all'utente di leggere in maniera più chiara un file XML, modificarlo e salvarlo per usi futuri.

MeXage

L'azienda in cui è stato svolto il presente lavoro di tesi e che mi ha permesso di entrare in contatto con il caso di studio concreto si chiama MeXage s.r.l., un'azienda di Castel Maggiore che collabora con partner importanti come:

- Alstom s.p.a, un gruppo industriale francese che opera nel settore della costruzione di treni e infrastrutture ferroviarie, leader di mercato nei sistemi di supervisione per il traffico ferroviario.
- FAAC s.p.a, un'azienda multinazionale italiana, specializzata in automazioni per cancelli e barriere, ingressi e porte automatiche, parcheggi e controllo accessi.
- Pietro Fiorentini s.p.a, leader nella fornitura di soluzioni per la distribuzione di gas e olio.

MeXage è costantemente impegnata in progetti che spaziano dalla definizione di sistemi, allo sviluppo software in ambienti desktop, web e mobile, fino al design di schede elettroniche basate su microcontrollori o PC embedded. [0]

Capitolo 1 - Meta Programmazione e CAD

1.1 Definizione Meta Programmazione

La meta programmazione è una tecnica di programmazione in cui i programmi hanno la capacità di trattare altri programmi come loro dati. Significa che un programma può essere progettato per generare, analizzare, leggere o trasformare altri programmi e persino modificarsi durante l'esecuzione. La meta programmazione riguarda la scrittura di codice che scrive codice.

Immaginiamo un ragazzo che costruisce automobili. Diciamo che sia la stessa cosa nell'utilizzare un computer. Ad un certo punto si rende conto che fa sempre la stessa cosa, più o meno. Quindi costruisce fabbriche per costruire automobili, ciò è molto efficiente. Ora sta programmando!

Tuttavia, ancora una volta, ad un certo punto, si rende conto di fare sempre la stessa cosa, in una certa misura. Ora decide di costruire fabbriche che costruiscono fabbriche che costruiscono automobili. Questa è meta programmazione. La meta programmazione consente di automatizzare attività di programmazione soggette a errori o ripetitive. Può essere usata per pre-generare tabelle di dati, per generare automaticamente codice standard o anche per testare l'ingegnoscità del programmatore nella scrittura di codice autoreplicante.

La meta programmazione non è solo generare codice, è programmare con i programmi. È possibile creare codice che crei codice, ispezioni il codice o lo modifichi effettivamente.

Nei linguaggi moderni, la meta programmazione è rappresentata in modi diversi. I modelli in C++ sono una forma di meta programmazione in fase di compilazione (compile time). Altri linguaggi utilizzano eval, goto, la programmazione automatica e altre tecniche per offrire funzionalità di meta programmazione in fase di esecuzione (runtime). Una classe di quei linguaggi che incoraggiano particolarmente la meta programmazione sono i linguaggi dinamici. Tali linguaggi sono generalmente tipizzati dinamicamente e in generale sono molto flessibili. Ruby e Python sono buoni esempi di questa classe.

La meta programmazione è immensamente potente, ma un problema tecnico nel sistema fa sì che tutti i vantaggi si trasformino in difficoltà spaventose. Quindi padroneggiarla non è scontato.

Una volta acquisita familiarità con le tecniche, la meta programmazione non è così complicata come potrebbe sembrare inizialmente. [1]

In altre parole, la programmazione è la capacità di istruire una macchina per eseguire attività passo dopo passo in modo tale da risolvere un problema. Abbiamo incaricato una macchina di questo perché anche se il problema iniziale può sembrare troppo complesso da risolvere per una macchina, i singoli passaggi non sono troppo complessi e possono essere risolti da una macchina, in modo più veloce e più facile che da un essere umano. Questo processo si occupa dei compiti facilmente prevedibili e ripetitivi in modo che l'essere umano possa concentrarsi sui compiti meno facilmente prevedibili e più complessi. Dopotutto, l'essere umano comprende l'intero quadro e a cosa equivalgono quei passaggi separati, la macchina no.

In altri termini è come fare un intervento chirurgico su un paziente e avere il paziente che ti aiuta a farlo.

Molte persone dicono che la meta programmazione non è così usata oggi, ma non è del tutto vero. La meta programmazione è oggi meno ovvia a causa degli strumenti altamente automatizzati con cui lavoriamo fuori dagli schemi. Evidenziazione del codice e analisi del codice sono esempi di codice che tratta altro codice come dato. Non è una meta programmazione molto sofisticata, ma è comunque meta programmazione. Con la frustrazione di dover costruire da solo tali sistemi da zero, il tipico sviluppatore di oggi non si accorge nemmeno che tipo di meta programma sta portando avanti anche perché ci sono molti strumenti che consentono agli sviluppatori di avere una vita facile.

Perché è utile per uno sviluppatore software sapere questo? Perché se è in grado di scrivere codice semplice da eseguire per una macchina e facile da capire per altri umani, allora ha raggiunto la perfezione. Ci sono molte persone che scrivono codice comprensibile che è altamente inefficiente e ci sono persone che scrivono codice altamente efficiente che è difficile da capire e mantenere (ovvero codice da buttare via). La comprensione umana del codice dipende dalla persona che lo legge. Imparare la meta programmazione aiuta a capire l'altro lato, il lato macchina della comprensione del codice. [2]

1.2 Diversi tipi di approccio alla Meta Programmazione

La meta programmazione permette ai programmatori di sviluppare programmi e scrivere codice che fa parte del modello di programmazione generica. Avere il linguaggio di programmazione stesso come tipo di dato di una classe (come ad esempio in Prolog, SNOBOL o Lisp) risulta particolarmente vantaggioso. Questo prende il nome di omoiconicità. La programmazione generica invoca una funzione di meta programmazione all'interno di un linguaggio consentendo di scrivere codice senza doversi preoccupare di specificare i tipi di dati dal momento che questi ultimi possono essere forniti come parametri quando vengono utilizzati.

La meta programmazione può essere suddivisa in tre approcci:

- Il primo approccio consiste nel presentare le parti interne del motore di runtime al codice di programmazione tramite API (Application Programming Interface) come quella di .NET per sistemi operativi Windows.
- Il secondo approccio è l'esecuzione dinamica di espressioni che contengono comandi di programmazione, frequentemente composti da stringhe, ma che possono anche provenire da altri metodi, come Javascript, che utilizzano argomenti o contesto. Di conseguenza, i programmi possono scrivere programmi. Anche se entrambi gli approcci possano essere utilizzati nella stessa lingua, la maggioranza delle lingue tende a prediligere per l'una o l'altra.
- Il terzo approccio consiste nell'uscire completamente dalla lingua. I sistemi di trasformazione dei programmi generici, come ad esempio i compilatori, accettando descrizioni linguistiche ed eseguendo trasformazioni arbitrarie su tali linguaggi, sono esempi concreti di implementazione della meta programmazione generale. Questo permette di usufruire della meta programmazione essenzialmente in qualsiasi lingua di destinazione, a prescindere dal fatto che tale lingua di destinazione abbia capacità di meta programmazione proprie. [3]

1.3 CAD

La progettazione assistita da computer è un modo per creare digitalmente disegni 2D e modelli 3D di prodotti reali, prima ancora che vengano realizzati.

CAD è l'abbreviazione di Computer-Aided Design ed è noto anche come CADD (Computer-Aided Design and Drafting). Si tratta di una tecnologia che consente di progettare e poi creare la relativa documentazione tecnica, sostituendo il disegno manuale con un processo automatizzato.

Persone che lavorano come architetti, progettisti, disegnatori e ingegneri, probabilmente avranno già usato programmi per disegnare CAD 2D o 3D, come AutoCAD, TinkerCAD e Creo. Questi rinomati software consentono di generare la documentazione relativa a un progetto, analizzare le idee progettuali, visualizzare i concetti tramite rendering fotorealistici e simulare la realizzazione di un progetto nel mondo reale.

1.3.1 CADdy

CADdy è un CAD 2D, software proprietario di MeXage. Esso è sempre in continua evoluzione e in continuo perfezionamento grazie al lavoro di sempre più menti diverse che aggiungono le loro migliori idee al progetto.

Il software è stato impiegato maggiormente in progetti che garantiscono la sicurezza della circolazione ferroviaria italiana sull'intera rete attualmente gestita da RFI (Rete Ferroviaria Italiana).

Il comportamento dei treni all'interno delle stazioni, che siano di ingresso, uscita o passaggio in esse, deve essere regolamentato ed attentamente studiato: prima RFI manteneva ogni informazione aggiornata su risorse cartacee. Ogni stazione veniva rappresentata attraverso il suo piano schematico, ovvero un disegno semplificato del percorso dei binari ed una serie di icone che simboleggiano oggetti di campo (Segnali, Passaggi a livello, etc.).

MeXage ha ricevuto l'incarico di fornire un'applicazione desktop che permettesse di disegnare, esportare e conservare piani schematici in forma digitale, e che fosse in grado, a partire da questi, di automatizzare la generazione delle condizioni di interscambio tra itinerari. Il gruppo

di sviluppo ha dunque provveduto a realizzare un software chiamato “CTS”, che prevedeva la possibilità di disegnare piani schematici di stazioni ferroviarie, grazie a funzionalità ereditate da CADdy, e di basare su di esse la costruzione degli itinerari dei treni. Le condizioni di sicurezza richieste per la formazione, il bloccamento e la liberazione degli itinerari, per la manovra dei segnali sono contenute all’interno di un documento di proprietà RFI intitolato “Tabella delle Condizioni”.

CADdy è stato impiegato in numerosi altri progetti come, ad esempio:

- *Alstom Multiview Draw*: un software di grafica vettoriale 2D per configurare l'interfaccia utente dei sistemi di supervisione e controllo del traffico ferroviario certificati SIL4 (sistemi sicuri). Lo strumento consente di definire una libreria di simboli grafici ed utilizzarla per disegnare il layout delle stazioni ferroviarie, anche automaticamente sfruttando degli algoritmi di piazzamento intelligente. L'applicativo supporta diversi formati, raster e vettoriali. La soluzione è utilizzata nei siti Alstom in Italia, Danimarca, India, Romania, e anche dai clienti finali.
- *Iconis Configuration Suite*: un pacchetto di applicativi per configurare il sistema di supervisione del traffico ferroviario sviluppato da Alstom Ferroviaria S.p.A.. La suite offre un'interfaccia grafica user friendly in grado di migliorare l'efficienza del processo produttivo. L'architettura della suite si basa su un meccanismo di plugin e di scripting con editor integrato che consente di adattarla alle esigenze specifiche dei vari utilizzi. È utilizzata da utenti in diversi paesi del mondo: Italia, Francia, Svezia, Danimarca, India, Romania.

Capitolo 2 - Pianificazione e Fasi Iniziali

2.1 Pianificazione del progetto

Avendo chiaro l'obiettivo finale del progetto si può passare alla sua pianificazione, questa parte è fondamentale e alla base di ogni progetto. Senza una struttura solida si sa già di intraprendere un percorso impraticabile, il progetto non partirà mai con il piede giusto, né si svilupperà nel modo previsto in origine. Una buona pianificazione si basa sui seguenti tre concetti principali:

1. *Strategia da utilizzare*: un momento durante il quale si deve fare il punto della situazione, riordinando le idee sulla base di quanto acquisito grazie alla ricerca e all'analisi in base agli obiettivi e alla pianificazione. Una fase sensibile che porta a realizzare il progetto, attraverso diverse idee, magari non tutte sempre applicabili e realizzabili, con lo scopo di portare il necessario valore aggiunto alla strategia di azione.
2. *Analisi degli incarichi*: tutte le varie responsabilità che costituiscono un progetto, il quale andrebbe diviso in sottoprogetti in modo tale da essere in grado di dedicare maggiori attenzioni ad ognuno di essi e per non rischiare di perdersi nell'analisi di un solo grande progetto.
3. *Cura dei dettagli*: deve sempre essere possibile avere una panoramica generale e, solo dentro ad ogni sottoprogetto, scendere nei dettagli per valutare cosa sia possibile aggiungere al progetto per una più chiara e semplice interpretazione.

Esistono dei punti fondamentali che devono essere presenti in un progetto, in primis una costante, una completa, precisa e chiara documentazione sia delle risorse utilizzate che delle attività svolte. Occorre determinare una scadenza del progetto, dividendo così i compiti da svolgere, per avere chiari gli obiettivi di tutti i sotto progetti. Non va trascurata logicamente la fase di unione dei singoli componenti, da non dare per scontata perché potrebbe nascondere insidie. È buona norma, perciò, preparare dei test in modo da poter verificare il comportamento.

2.2 Suddivisione in sottoprogetti

Analizzando il progetto del SymbolEditor nella sua interezza si è deciso quindi di dividerlo in compiti minori da dover svolgere in maniera progressiva. L'ordine con cui verranno illustrati i singoli sottoprogetti seguirà l'interazione che un utente avrebbe con il programma.

Il lavoro è stato suddiviso in diversi task presenti nella sezione del Backlog di Azure DevOps dedicata al progetto. Ogni task da completare corrispondeva ad una funzione specifica del plugin. Appena si iniziava un task bisognava modificare lo stato da "New" ad "Active". Non appena finito di completare il task lo stato veniva modificato in "To be Tested" così da consentire al tutor aziendale di controllare il lavoro svolto e, in caso di nessun errore rilevato, permettere a quest'ultimo di modificare lo stato su "Closed". Ad avvenuto completamento dei vari task era necessario fare il push del progetto così da avere sempre la versione più aggiornata su cui continuare a lavorare.

2.3 Fasi iniziali del progetto

Come prima cosa è stato necessario studiare come fosse strutturato un file XML per avere più padronanza e saperlo usare per la creazione di un template che sarebbe stato il punto di partenza di tutto il processo conversione. È stato necessario fare delle prove per convertire un file XML in un file CS (file di codice sorgente scritto in C#). Nella classe "Program.cs" sono stati implementati i metodi per la creazione della cartella di destinazione del zip file GA.

Affinché si possa avere una DLL (Dynamic-link library) si è creato un progetto libreria su Visual Studio, eliminando la classe predefinita "Class1.cs", copiando il file CS nella cartella del progetto e facendo la build del progetto così da avere un file con estensione DLL.

Una DLL, libreria di collegamento dinamico in italiano, indica una libreria software che viene caricata dinamicamente in fase di esecuzione, invece di essere collegata staticamente ad un eseguibile in fase di compilazione.

In parole più semplici, è una raccolta di piccoli programmi che i programmi più grandi possono caricare quando necessario per completare attività specifiche. Il piccolo programma, chiamato file DLL, contiene istruzioni che aiutano il programma più grande a gestire quella che potrebbe non essere una funzione principale del programma originale.

Dato che tutte queste operazioni vanno fatte in automatico da riga di comando si è sfruttata la classe *ProcessStartInfo*, la quale specifica un insieme di valori usati all'avvio di un processo, e *ProcessStart* la quale avvia una risorsa di processo e la associa ad un componente *Process*.

ProcessStartInfo viene usata insieme al componente *Process*. Quando si avvia un processo usando la classe *Process*, è possibile accedere alle informazioni di elaborazione oltre a quelle disponibili durante il collegamento ad un processo in esecuzione.

È possibile usare la classe *ProcessStartInfo* per un migliore controllo sul processo avviato. È necessario impostare almeno la proprietà *FileName* manualmente o usando il costruttore. Il nome del file è un'applicazione o un documento. In questo caso viene definito un documento come qualsiasi tipo di file a cui è associata un'azione.

Inoltre, è possibile specificare i valori *Arguments* delle proprietà come argomenti della riga di comando da passare alla routine di apertura del file. Ad esempio, se si specifica un editor di testo nella proprietà *FileName*, è possibile utilizzare la proprietà *Arguments* per specificare un file di testo da far aprire all'editor.

Lo standard input è in genere la tastiera e lo standard output e lo standard error sono in genere lo schermo del monitor. Tuttavia, è possibile usare le proprietà *RedirectStandardInput*, *RedirectStandardOutput* e *RedirectStandardError* per fare in modo che il processo possa ottenere l'input o restituire l'output su un file o su un altro dispositivo. Se si usano le proprietà *StandardInput*, *StandardOutput* o *StandardError* nel componente *Process*, è necessario innanzitutto impostare il valore corrispondente nella proprietà *ProcessStartInfo*. In caso contrario, il sistema genera un'eccezione.

Si può impostare la proprietà *UseShellExecute* per specificare se avviare il processo usando la shell del sistema operativo. Se *UseShellExecute* è impostato su false, il nuovo processo eredita rispettivamente i flussi di standard input, standard output e di standard error del processo chiamante, a meno che le proprietà *RedirectStandardInput*, *RedirectStandardOutput* o *RedirectStandardError* siano impostate rispettivamente su “true”. È possibile modificare il valore di qualsiasi proprietà *ProcessStartInfo* fino all'avvio del processo. Dopo aver avviato il processo, la modifica di questi valori non ha alcun effetto. [4]

ProcessStartInfo è stata di fondamentale aiuto per tutte le volte che è stato necessario fare operazioni da riga di comando. Nell'immagine sotto riportata è presente un esempio di utilizzo di *ProcessStartInfo* per creare un nuovo progetto libreria in Visual Studio specificando il path assoluto del progetto da creare.

```
// CREA IL .csproj
1 riferimento
private static bool CreateProject(string pathProj)
{
    var CDMcreateProj = @"new classlib --output {pathProj}";
    ProcessStartInfo startInfoCreateProj = new()
    {
        CreateNoWindow = false,
        UseShellExecute = true,
        FileName = @"dotnet.exe",
        WindowStyle = ProcessWindowStyle.Hidden,
        Arguments = CDMcreateProj
    };

    try
    {
        using var exeProcess = Process.Start(startInfoCreateProj);
        exeProcess.WaitForExit();
    }
    catch (IOException iox)
    {
        _Log.Error(iox.Message);
        return false;
    }
    return true;
}
```

Figura 2 – Esempio utilizzo *ProcessStartInfo*

Capitolo 3 -

Template XML, NuGet Packages e CommandLineParser

3.1 Template XML per l'estrazione dei dati

Si è deciso di usare come template per l'estrazione dei dati un file XML per una rappresentazione più trasparente possibile, una libertà di espressione non vincolata dall'uso di tag in numero limitato (come accade per documenti HTML) ed una semplicità di lettura anche da parte di persone non altamente specializzate nell'ambito informatico.

Il linguaggio XML (eXtensible Markup Language) è ampiamente diffuso e sono ben noti i motivi per cui è suggerito usarlo, in particolare per la potenza espressiva dei markup. Il markup è il processo di impiego di elementi denominati tag (o anche token) per definire l'aspetto visivo, la struttura e, nel caso di XML, il significato di qualsiasi dato. È possibile vedere che i tag XML rendono possibile conoscere esattamente il tipo di dato che si sta esaminando.

È importante però che un file XML sia ben formato, conforme ad un set di regole molto rigide che regolano l'XML. Se un file non è conforme a queste regole, l'XML smette di funzionare. Per esempio, ad ogni tag di apertura deve corrispondere un tag di chiusura. Se si rimuove un tag e si prova ad aprire il file, l'applicazione impedirà di usare il file e verrà visualizzato un messaggio di errore.

Pur non rappresentando un vincolo, la conoscenza di regole per la creazione di documenti XML ben formati, anche se facili da capire, è fondamentale tenere presente che i dati XML possono essere condivisi tra applicazioni e sistemi solo se sono ben formati. Se non è possibile aprire un file XML, è altamente probabile che non sia ben formato.

Un altro punto di forza del linguaggio XML è il fatto di essere indipendente dalla piattaforma, ossia qualsiasi applicazione progettata per usarlo consente di leggere ed elaborare dati XML, indipendentemente dall'hardware o dal sistema operativo. Con i tag XML appropriati è ad esempio possibile usare un'applicazione per PC per aprire e gestire i dati provenienti da un mainframe. Indipendentemente dall'autore di un testo con dati XML, è in aggiunta possibile usare gli stessi dati in diverse applicazioni. Grazie alla portabilità, l'XML è ormai una delle tecnologie più diffuse per lo scambio di dati tra database e personal computer. [5]

La struttura finale del template del file XML che sia comprensibile ed allo stesso tempo efficiente è la seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin Name="LogicGate" Description="PluginDescriptionVersion1" Version="1.0.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <PackageSource>
    <SourceItem Folder="programmi_pc\Paket\packages"/>
    <SourceItem Folder="DLL_ToUse\myLogicNuguet"/>
  </PackageSource>
  <PackageMap>
    <PackageItem Name="MexageCoreLib" Nuget="Microsoft.Azure.DocumentDB.Core" Version="1.0.0"/>
    <PackageItem Name="MexageExtensionLib" Nuget="Newtonsoft.Json.Bson" Version="1.0.2"/>
  </PackageMap>
  <UsingMap>
    <MapItem ClassName="LocDisplayName" Using="MexageCoreLib.attribute" Package="MexageCoreLib"/>
    <MapItem ClassName="LocDescription" Using="MexageCommonInterfaces.project" Package="MexageStandardLib"/>
    <MapItem ClassName="LocDescription2" Using="MexageExtensionLib.natives" Package="MexageExtensionLib"/>
    <MapItem ClassName="LocDescription3" Using="MexageCoreLib.attribute" Package="MexageCoreLib"/>
  </UsingMap>
</plugin>
```

Figura 3 – Esempio struttura parte iniziale template XML

In questa prima parte del file si specificano i path relativi ai Package Sources (pacchetti dei sorgenti) che dovranno essere aggiunti a quelli già presenti di default. Essi sono presenti in locale, quindi è possibile importare nel proprio progetto dei pacchetti sviluppati dall'utente stesso e non presenti online nella NuGet Gallery. I Package Sources forniscono tutti i file necessari per la compilazione o altrimenti per creare la parte desiderata di software. In *PackageMap* abbiamo la lista dei *PackageItem*, ovvero gli effettivi pacchetti NuGet che dovranno essere scaricati ed importati nel progetto.

NuGet è un gestore di pacchetti che fornisce codice sorgente compilato (DLL) e altri file (script e immagini) relativi al codice. Un pacchetto NuGet assume la forma di un file zip con estensione “.nupkg”. Ciò semplifica l'aggiunta, l'aggiornamento e la rimozione di librerie nelle applicazioni sviluppate in Visual Studio. L'uso di NuGet per installare e aggiornare i pacchetti riduce il lavoro manuale di configurazione di librerie di terze parti in un'applicazione estraendo un file zip e aggiungendo gli assembly necessari a riferimenti e file.

La parte centrale del file è quella di maggior interesse in quanto rappresenta il cardine del progetto: i tag inerenti alla creazione delle classi. Ogni classe presenta:

- nome
- visibilità

- la classe che estende (se ne estende qualcuna)
- descrizione
- attributo di classe

Possiamo osservare che i field sono composti da:

- nome
- tipo
- descrizione

Ogni field può presentare diversi attributi, ognuno dei quali presenta:

- nome
- parametri

Le proprietà di classe sono composte nel seguente modo:

- nome
- tipo
- descrizione

Proseguendo nell'analisi del file XML notiamo i tag *MapView* formati da una lista di *MapViewItem*, i quali serviranno per formare il file "map.json" contenente gli SVG. Essi sono composti da:

- valore
- descrizione
- lista di *ViewProperty* (a loro volta composti da nome e valore)

I *MapViewItem*, come il nome suggerisce, serviranno per la parte grafica. In particolare, l'attributo *Value* specifica quale preciso file SVG dal pool di path specificati nel file XML (i tag inerenti agli SVG saranno osservati nella parte successiva) selezionare per quel preciso oggetto.

```
<Type Name="And" Visibility="public" extend="LogicGate" Description="ClassDescriptionAND">
  <Field Name="PropertyNameAND" Type="FieldTypeAND" Description="FieldDescriptionAND">
    <Attribute Name="AND_Field_Name1" Parameters="ParamField1, ParamField2, ....., ParamFieldN"/>
    <Attribute Name="AND_Field_Name2" Parameters="ParamField1, ParamField2, ....., ParamFieldN"/>
  </Field>
  <Property Name="NInput" Type="int" Description="Number of signals in input">
    <Attribute Name="LocDescription" Parameters="Loc_LogicGate_NInput"/>
  </Property>
  <Attribute Name="AND_Class_Name1" Parameters="ParamClass1, ParamClass2, ....., ParamClassN"/>
  <MapView>
    <MapViewItem Value="and2.svg" Description="LogicGateAND con due input">
      <ViewProperty Name="NumberOfInput" Value="2" />
      <ViewProperty Name="NumberOfOutput" Value="1" />
    </MapViewItem>
    <MapViewItem Value="and3.svg" Description="LogicGateAND con tre input">
      <ViewProperty Name="NumberOfInput" Value="3" />
      <ViewProperty Name="NumberOfOutput" Value="1" />
    </MapViewItem>
  </MapView>
</Type>
```

Figura 4 – Esempio struttura parte centrale template XML

L'ultima parte del file XML è dedicata agli *Enum* e agli *SVG*. Questi ultimi sono identificati dal nome e dal path relativo al file XML dove sono collocati. Un esempio di enum è l'orientamento dell'oggetto, ma è possibile specificare illimitati tipi di enum per arricchire ulteriormente la specificità dell'oggetto. Tramite il tag *svg* si specificano i path relativi a tutti gli SVG (Scalable Vector Graphics) che saranno utilizzati. Si è deciso di fare uso dei file SVG in quanto si è in grado di visualizzare oggetti di grafica vettoriale e quindi di salvare immagini in modo che si possano ingrandire e rimpicciolire a piacere senza perdere la risoluzione grafica. A differenza dei file raster basati su pixel, come i jpeg ad esempio, i file vettoriali memorizzano le immagini tramite formule matematiche basate su punti e linee su una griglia.

```
<Enum Name="EDirection" Description="Direction of objects" Values="Undef, Left, Right, Top, Bottom"/>
<svg>
  <svgItem Name="and2.svg" Path="svgAnd\and2.svg"/>
  <svgItem Name="and3.svg" Path="svg2\and3.svg"/>
</svg>
</plugin>
```

Figura 5 – Esempio struttura parte finale template XML

3.2 Import Package Source e NuGet Packages

Per aggiungere nuovi Package Sources a quelli già presenti di default è stato implementato il metodo *FindUsingLibs* il quale riunisce in una lista di elementi distinti tutti i nomi degli attributi di cui il file CS necessita, va a cercare in *UsingMap* il corrispettivo *ClassName* in *MapItem* e aggiunge i *Package Name* in una lista che va confrontata con gli elementi *PackageItem* presenti in *PackageSource*. Ciò permette di importare nel progetto solo i pacchetti strettamente necessari al funzionamento del componente che si vuole creare, riducendo al minimo le risorse utilizzate. Nella figura mostrata di seguito è stata omessa la parte finale di popolamento della lista dei Package consistente in due istruzioni di foreach, il primo, chiamante il metodo *AddNugetSource* ad ogni ciclo, mentre il secondo chiamante il metodo *ImportPacket* ad ogni ciclo.

```

1 riferimento
public string FindUsingLibs(PluginDescriptor plugin)
{
    UsingMapDescriptor usingMap = plugin.UsingMap;
    PackageMapDescriptor packageMap = plugin.PackageMap;

    HashSet<string> attrName = new HashSet<string>();

    var list = plugin.Classes.SelectMany(c => c.Attributes
                                                .Union(c.Properties.SelectMany(p => p.Attributes))
                                                .Union(c.Fields.SelectMany(f => f.Attributes)))
                            .Select(a => a.Name)
                            .Distinct();

    //lista di Attribute names delle classi da importare
    var mapList = list.Union(usingMap.Items.Select(c => c.ClassName))
                    .Where(a => ((list.Contains(a) &&
                                usingMap.Items.Select(c => c.ClassName).Contains(a))));

    List<MapItemDescriptor> mapAttr = new List<MapItemDescriptor>();

    foreach (var item in usingMap.Items) //Class.Fields
    {
        foreach (var l in mapList)
        {
            if (item.ClassName.Equals(l))
            {
                mapAttr.Add(item);
            }
        }
    }
}

```

Figura 6 – Parte iniziale metodo FindUsingLibs

Avendo la lista dei Package Sources di cui il progetto necessita si è deciso di implementare i metodi *AddNugetSource* per aggiungere i nuovi Package Sources e *ImportPacket* per importare nel progetto i NuGet Packages.

```

1 riferimento
public void AddNugetSource(SourceItemDescriptor sourceItem)
{
    var sourceFolder = $"{sourceItem.Folder}";
    var fullPath = Path.Combine(Path.GetDirectoryName(this.myXMLfile), sourceFolder);

    // AGGIUNGE NUGET FOLDERS
    var CDMaddNuget = @"$nuget add source {fullPath}\\";
    ProcessStartInfo startInfo = new()
    {
        CreateNoWindow = false,
        UseShellExecute = true,
        FileName = @"dotnet.exe",
        WindowStyle = ProcessWindowStyle.Hidden,
        Arguments = CDMaddNuget
    };
    try
    {
        using var exeProcess = Process.Start(startInfo);
        exeProcess.WaitForExit();
    }
    catch (IOException iox)
    {
        _Log.Error($"Add Nuget Source {iox.Message}");
    }
}

```

Figura 7 – Metodo AddNugetSource

È utile sottolineare l'utilizzo del metodo *Path.Combine* per ottenere il path assoluto del file XML in base al *SourceFolder*. *Path.Combine* è progettato per concatenare stringhe in una sola che rappresenta il percorso di un file. Tuttavia, se un argomento diverso dal primo contiene un percorso "rooted", tutti i componenti di percorso precedenti vengono ignorati e la stringa restituita inizia con il componente del percorso "rooted".

```
1 riferimento
public void ImportPacket(PackageItemDescriptor packageItem, string pathProj)
{
    if (!string.IsNullOrEmpty(packageItem.Nuget))
    {
        var packageName = packageItem.Nuget;
        var packageVersion = packageItem.Version;

        // IMPORTA PACKAGE DA NUGET NEL PROGETTO
        var CDMaddNuget = @$"add {pathProj} package {packageName} -v {packageVersion}";
        ProcessStartInfo startInfo = new()
        {
            CreateNoWindow = false,
            UseShellExecute = true,
            FileName = @"dotnet.exe",
            WindowStyle = ProcessWindowStyle.Hidden,
            Arguments = CDMaddNuget
        };
        try
        {
            using var exeProcess = Process.Start(startInfo);
            exeProcess.WaitForExit();
        }
        catch (IOException iox)
        {
            _Log.Error($"Import Packet {iox.Message}");
        }
    }
}
```

Figura 8 – Metodo *ImportPacket*

Possiamo notare come ancora una volta sia stata di fondamentale importanza la classe *ProcessStartInfo*. Scrivere codice testato e funzionante consente ai programmatori di riutilizzare lo stesso codice per funzionalità simili in più app, riducendo il tempo necessario per sviluppare nuove applicazioni. È evidente come, con delle semplici modifiche dei valori di *Arguments*, sia possibile ridurre notevolmente il tempo di scrittura e testing del codice (questa ultima parte è infatti praticamente azzerata). Il riutilizzo di software non solo migliora la produttività, ma ha anche un impatto positivo sulla qualità e la manutenibilità dei prodotti software. Lo sviluppo di software veloce, affidabile e sicuro richiede costantemente molte capacità e conoscenze di programmazione. Pertanto, gli sviluppatori devono analizzare le condizioni e i requisiti di un'applicazione prima di procedere con il riutilizzo del codice.

3.3 Inserimento dati da parte dell'utente

Per rendere più piacevole l'esperienza utente ci si è avvalsi della libreria *CommandLineParser*. La libreria del Parser da riga di comando offre alle applicazioni CLR un'API chiara e concisa per manipolare gli argomenti della riga di comando e le attività correlate, come la definizione di opzioni e verb commands. Consente di visualizzare una schermata di aiuto con un elevato grado di personalizzazione e un modo semplice per segnalare errori di sintassi all'utente finale. Come opzioni per il corretto utilizzo del parser abbiamo deciso di creare le seguenti:

-o : specificare il path del progetto libreria da creare.

-d : specificare il path della cartella dove inserire i file DLL, manifest.json, maps.json e la cartella degli SVG.

-x : specificare il path del file XML.

```
//--- PARSE CLI ARGUMENTS ---\\
if ((Parser.Default.ParseArguments<CommandLineOption>(args) is not Parsed<CommandLineOption> parsedOptions))
{
    // SOMETHING WENT WRONG
    _Log.Error("Invalid arguments. Parsing has errors...");
    return;
}
else
{
    var myXMLfile = String.Empty;
    var dllPath = String.Empty;
    var pathProj = String.Empty;

    var result = Parser.Default.ParseArguments<CommandLineOption>(args)
        .WithParsed<CommandLineOption>(o =>
        {
            if (o.XMLPath is not null)
            {
                myXMLfile = o.XMLPath;
                _Log.Info(myXMLfile);
            }
            if (o.DllOutputDir is not null)
            {
                dllPath = o.DllOutputDir;
                _Log.Info(dllPath);
            }
            if (o.ProjOutputDir is not null)
            {
                pathProj = o.ProjOutputDir;
                _Log.Info(pathProj);
            }
        }
    );
}
```

Figura 9 – Esempio d'uso della classe *CommandLineParser*

Capitolo 4 - Deserializzazione e Serializzazione del file XML

4.1 Deserializzazione del file XML

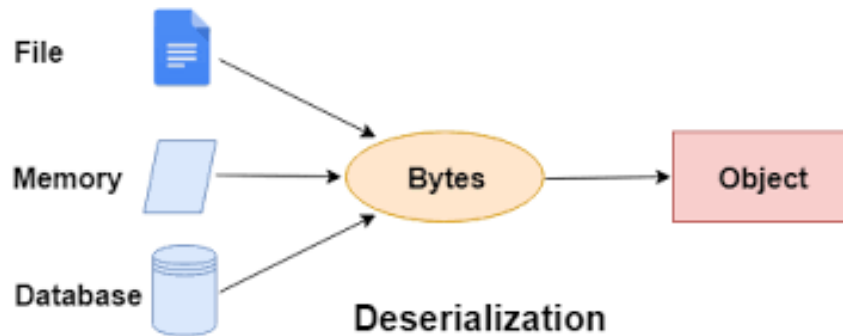


Figura 10 – Schema funzionamento Deserializzazione

Avendo chiara la struttura del file XML, il passo successivo da compiere è stato quello di deserializzare il file per estrarre i dati e memorizzarli per usarli successivamente. Una parte cruciale del progetto è stato il *Deserializer*, una classe che riceve il path di un file XML e lo serializza in un oggetto di tipo *PluginDescriptor*. La Deserializzazione è il processo di lettura di un documento XML e la costruzione di un oggetto fortemente tipizzato. Quando si deserializza un oggetto, il formato di trasporto determina se verrà creato uno stream oppure un object file. Una volta determinato il formato di trasporto, è possibile chiamare i metodi *Serialize* o *Deserialize*, in base alle necessità.

Per deserializzare un oggetto oppure un file i passi da compiere sono i seguenti:

1. Costruire un oggetto *XmlSerializer* che utilizza il tipo dell'oggetto da deserializzare (nel nostro caso di studio l'oggetto in questione è un *PluginDescriptor*)
2. Chiamare il metodo *Deserialize* per produrre una replica dell'oggetto. Quando si deserializza, è necessario eseguire il cast dell'oggetto restituito al tipo dell'originale che deserializza l'oggetto da un file (anche se potrebbe anche essere deserializzato da uno stream). [6]

```

1 riferimento
public Deserializer(string file)
{
    filePath = file;
    var serializer = new XmlSerializer(typeof(PluginDescriptor));
    using var myXML = new FileStream(filePath, FileMode.Open);
    {
        var tmp = serializer.Deserialize(myXML);

        if (tmp is PluginDescriptor plugin)
        {
            SetDefaultValue(plugin);
            Plugin = plugin;
        }
    }
}

```

Figura 11 – Classe Deserializer

Quando verrà chiamato il costruttore del *Deserializer* nel *Program.cs* avverrà automaticamente l'analisi del file XML specificato dall'utente e la creazione di un oggetto di tipo *PluginDescriptor*. A cascata verranno creati tutti gli altri oggetti (*ClassDescriptor*, *EnumDescriptor*, *FieldDescriptor*, *MapItemDescriptor*, *MapViewClassDescriptor*, *MapViewDescriptor*, *MapViewItemDescriptor*, *PackageItemDescriptor*, *PackageMapDescriptor*, *PackageSourceDescriptor*, *PropertyDescriptor*, *SourceItemDescriptor*, *SVGDescriptor*, *SVGItemDescriptor*, *UsingMapDescriptor* e *ViewPropertyDescriptor*).

4.2 Serializzazione del file XML

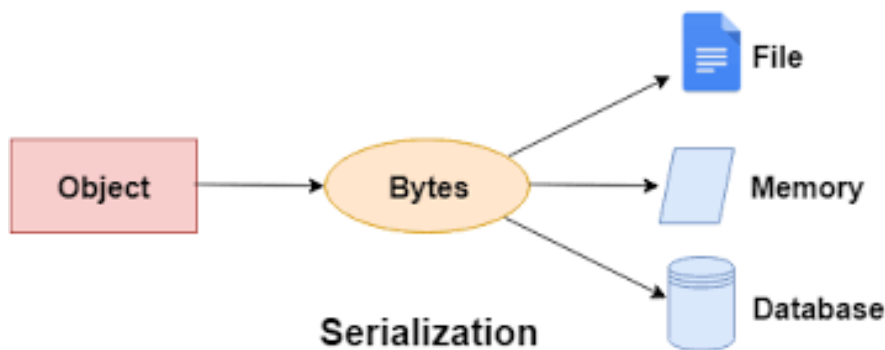


Figura 12 – Schema funzionamento Serializzazione

Adesso che abbiamo ottenuto un oggetto di tipo *PluginDescriptor*, il passo successivo da compiere è serializzarlo in un file di tipo CS, una classe contenente tutte le partial class presenti nel file XML. Ad esempio, prendendo come ambito applicativo i gate logici, il *PluginDescriptor* sarà rappresentato da "LogicGate" (esso sarà il namespace della classe) e

successivamente avremo tutte le partial class che potranno ad esempio essere i vari logic gate (AND, OR, NOT, XOR, NOR, NAND, XNOR).

La serializzazione è il processo inverso della Deserializzazione. Con il termine serializzazione indichiamo il processo di conversione di un oggetto in un flusso di byte, allo scopo di archiviare tale oggetto o trasmetterlo ad un file, ad una memoria o ad un database. La finalità principale della serializzazione è salvare lo stato di un oggetto per consentirne la sua ricreazione in caso di necessità. Questo oggetto viene serializzato in un flusso contenente i dati. È possibile che il flusso possa presentare informazioni sul tipo dell'oggetto, come ad esempio il nome dell'assembly e la versione. Da tale flusso, l'oggetto può essere archiviato in un file, una memoria oppure in un database.

La serializzazione permette di salvare lo stato di un oggetto e ricrearlo in base alle esigenze, fornendo l'archiviazione di oggetti e lo scambio di dati. Alcune azioni che uno sviluppatore può eseguire tramite la serializzazione sono:

- Invio dell'oggetto a un'applicazione remota tramite un servizio Web.
 - Passaggio di un oggetto da un dominio ad un altro.
 - Passaggio di un oggetto tramite un firewall come stringa JSON o XML.
 - Gestione della sicurezza o delle informazioni specifiche dell'utente nelle applicazioni.
- [7]

I punti che è necessario notare durante la creazione di una classe per la serializzazione sono i seguenti:

- La serializzazione XML serializza solo i campi e le proprietà pubblici.
- La serializzazione XML non include alcuna informazione sul tipo.
- Abbiamo bisogno di un costruttore predefinito/non parametrizzato per serializzare un oggetto.
- Le proprietà di sola lettura (ReadOnly) non sono serializzate.

La classe *XmlSerializer* (ubicata nel namespace *System.Xml.Serialization*) viene utilizzata per serializzare e deserializzare. Viene chiamato il metodo della classe *Serialize*. Dato che dobbiamo serializzare in un file, creiamo un oggetto *StreamWriter*. Dato che quest'ultimo implementa *IDisposable*, abbiamo usato "using" in modo da non dover chiudere il writer.

Ogni proprietà da serializzare presente nelle varie classi, per essere riconosciuto dal *Serializer*, deve essere contrassegnata da uno specifico attributo di proprietà. Gli attributi più comuni disponibili durante la serializzazione sono:

- *XmlAttribute*: questo membro verrà serializzato come attributo XML.
- *XmlElement*: il campo verrà serializzato come elemento XML.
- *XmlIgnore*: il campo verrà ignorato durante la serializzazione.
- *XmlRoot*: rappresenta l'elemento radice del documento XML.

Nel progetto si è ricorso solo agli attributi *XmlAttribute* per serializzare un singolo attributo di classe ed *XmlElement* quando una classe conteneva liste di altri oggetti. Ad esempio, *PluginDescriptor* presenta una descrizione (quindi in questo caso si è usato *XmlAttribute*) ma una lista di *ClassDescriptor* (quindi in questo caso si è usato *XmlElement*). Avendo una struttura fortemente innestata, gli attributi di serializzazione permettono, attraverso i tag, l'inserimento dei dati presi dal file nelle rispettive classi in automatico. È evidente quanto sia solido ed efficiente la serializzazione. Modificando la lunghezza del file di partenza, non bisogna modificare di conseguenza gli attributi di serializzazione o il codice in quanto il programma si adatterà nella gestione dei dati.

Il *Deserializer*, sfruttando il metodo *SerializeIntoCSProj*, crea un nuovo oggetto *CSharpExporter*, una classe creata appositamente per restituire una stringa, la quale avrà la struttura perfetta di una classe CS riconosciuta da "CADdy". Questa stringa andrà scritta su un file con estensione CS per poi essere copiata nel progetto creato precedentemente.

1 riferimento

```
public void SerializeIntoCSProj(string projPath)
{
    var csFile = "csFileToCopy.cs";
    var csContent = Plugin.Accept(new CSharpExporter(projPath, filePath));
    using (StreamWriter writetext = new StreamWriter(csFile))
    {
        writetext.Write(csContent);
    }
    _Log.Info("The cs file to put into .csproj has been created\n");
}
```

Figura 13 – Metodo *SerializeIntoCSProj*

4.3 Visitor Design Pattern

Affinché il programma sia scalabile si è sfruttato il *Visitor*, un modello di progettazione comportamentale che consente di aggiungere nuovi comportamenti alla gerarchia di classi esistente senza alterare il codice esistente.

Per la programmazione orientata agli oggetti, il pattern Visitor consente la definizione di una nuova operazione su una struttura ad oggetti senza cambiare le classi degli oggetti. Il prezzo da pagare è quello di stabilire in anticipo il set di classi, e ciascuna di esse deve avere un cosiddetto metodo *Accept*().

```
1 riferimento
public string Visit(PluginDescriptor plugin)
{
    var description = DescriptionSummary(plugin.Description);
    var usingString = FindUsingLibs(plugin);
    // using {usingString};

    return $"{@"
namespace {plugin.Name}
{{
    {description}
    {string.Join("\t\n", plugin.Enums.Select(cl => cl.Accept(this)))}
    {string.Join("\n", plugin.Classes.Select(cl => cl.Accept(this)))}
}}
";
}
```

Figura 14 – Esempio d'uso Visitor

Lo scopo principale del *Visitor* è quello di astrarre funzionalità che possono essere applicate ad una gerarchia aggregata di oggetti *Element*. L'approccio incoraggia la progettazione di classi *Element* “leggere” (nel nostro caso chiamato *CodeElement*), poiché la funzionalità di elaborazione viene rimossa dal loro elenco di responsabilità. Nuove funzionalità possono essere facilmente aggiunte alla gerarchia di ereditarietà originale creando una nuova sottoclasse *Visitor*.

Il corretto utilizzo del *Visitor* è il seguente: bisogna creare una gerarchia di classi *Visitor* che definisca un metodo puro *Visit*() nella classe base astratta per ogni classe derivata concreta

nella gerarchia del nodo aggregato. Ogni metodo *Visit()* accetta un singolo argomento: un puntatore o un riferimento a una classe derivata dall'*Element* originale.

Ogni operazione da supportare è modellata con una classe derivata concreta della gerarchia dei *Visitor*. I metodi *Visit()* dichiarati nella classe base *Visitor* sono ora definiti in ogni sottoclasse derivata.

Il metodo *Accept()* è definito per ricevere un singolo argomento, un puntatore o un riferimento alla classe base astratta della gerarchia dei *Visitor*.

Ogni classe derivata concreta della gerarchia *Element* implementa il metodo *Accept()* semplicemente chiamando il metodo *Visit()* sull'istanza concreta derivata della gerarchia *Visitor* che è stata passata, passando il suo puntatore "this" come unico argomento.

Quando si deve eseguire un'operazione, il programma crea un'istanza dell'oggetto *Visitor*, chiama il metodo *Accept()* su ciascun oggetto *Element* e passa l'oggetto *Visitor*.

Il metodo *Accept()* fa sì che il flusso di controllo trovi la sottoclasse *Element* corretta. Quindi, quando viene invocato il metodo *Visit()*, il flusso di controllo viene trasferito alla sottoclasse *Visitor* corretta.

Il modello *Visitor* semplifica l'aggiunta di nuove operazioni: basta aggiungere semplicemente una nuova classe derivata da *Visitor*. Tuttavia, se le sottoclassi nella gerarchia del nodo aggregato non sono stabili, mantenere sincronizzate le sottoclassi *Visitor* richiede uno sforzo proibitivo.

Un'obiezione sensata al modello *Visitor* è quella di una regressione alla scomposizione funzionale: separare gli algoritmi dalle strutture dati. Sebbene questa sia un'interpretazione legittima, forse una prospettiva migliore è rappresentata dall'obiettivo di promuovere il comportamento non tradizionale allo stato di oggetto. [8]

La gerarchia di *Element* è usata come "universal method adapter". L'implementazione del metodo *Accept()* in ogni classe derivata da *Element* è sempre la stessa. Esso non può essere spostato nella classe base *Element* ed ereditato da tutte le classi derivate perché un riferimento a "this" nella classe *Element* è sempre mappato al tipo base *Element*.

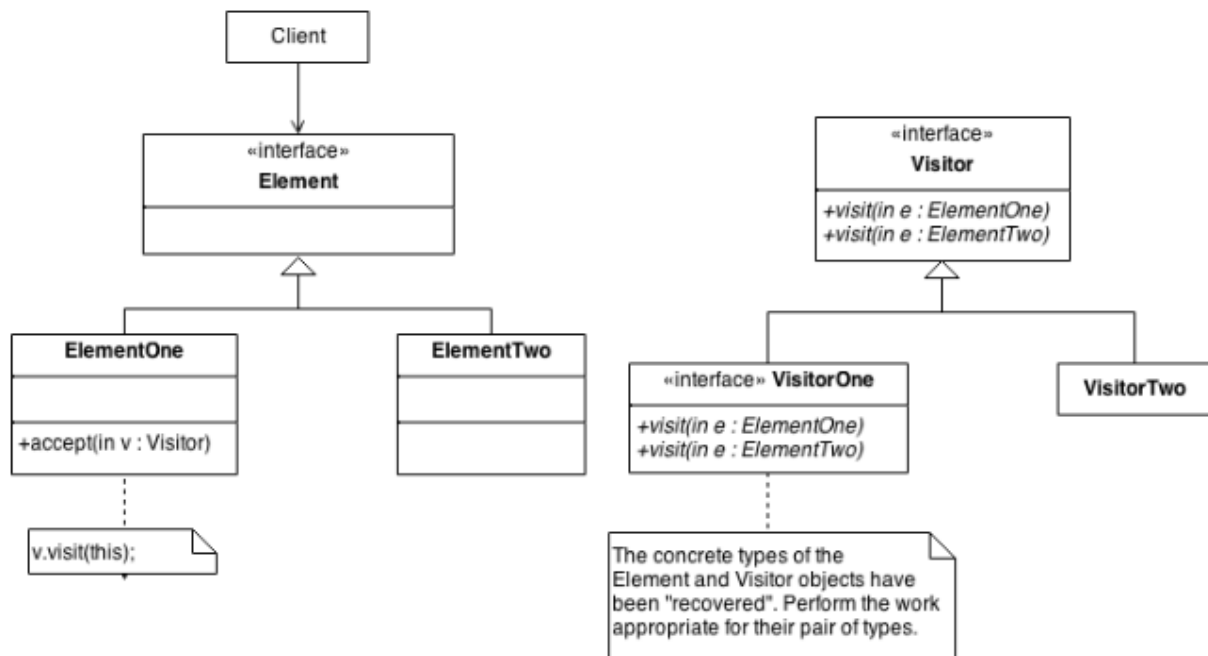


Figura 15 – Schema UML Visitor



Figura 16 – Classi che usano Visit

Nella figura riportata di fianco sono state evidenziate in giallo le classi che implementano il metodo *Visit*, mentre sono state evidenziate in verde tutte le classi che implementano l'interfaccia *IVisitorElement* (anche quelle in giallo fanno parte di questo gruppo). In questo progetto *IVisitorElement* rappresenta il generico elemento di un *Visitor* al cui interno è implementato il metodo *Accept* generico. Come si può osservare, la cartella *dto* (Data Transfer Object) è composta principalmente dalle classi usate nel Visitor Design Pattern.

È consigliato l'utilizzo del Visitor Design Pattern quando:

- Vogliamo separare i comportamenti indipendenti e non correlati dalla classe del tipo in un'altra classe e vogliamo cambiare i comportamenti in modo dinamico senza modificare il codice delle classi del tipo.
- Vogliamo compiere un tipo di operazione simile su oggetti di tipo diverso raggruppati in collection o gerarchia.
- Abbiamo una gerarchia degli oggetti nota a priori e con bassa plausibilità di modifiche, ma al contrario c'è una forte probabilità di aggiunta di nuove operazioni in futuro. Dato che questo pattern ci consente di separare l'operazione dalla struttura dell'oggetto, ci risulta molto facile aggiungere nuove operazioni sotto forma di Visitor. C'è sempre da tenere a mente però che questo funzionerà finché la struttura dell'oggetto rimarrà invariata.

Il Visitor Design Pattern risulta invece poco utilizzabile nelle seguenti condizioni:

- Il Visitor Pattern richiede che gli argomenti e il tipo restituito per i metodi di “visita” debbano essere noti in fase di progettazione. Risulta facile capire quindi che questo modello non è adatto nella situazione in cui i tipi vengono modificati di frequente, perché una volta introdotto il nuovo tipo, tutto il Visitor dovrà essere modificato di conseguenza.
- Quando i comportamenti sono correlati non all'intera gerarchia bensì alla singola istanza dell'oggetto. Tali comportamenti non dovrebbero essere implementati tramite il *Visitor*, poiché il *Visitor* viene utilizzato per definire comportamenti che verranno applicati all'intera gerarchia. [9]

Capitolo 5 - Creazione zip GA e log

5.1 Serializzazione JSON files e creazione ZIP GA



Figura 17 – Schema file contenuti nella cartella compressa

Il passo successivo, dopo aver compilato il progetto ed ottenuto un DLL (anche in questo caso è stata fondamentale la classe *ProcessStartInfo*), è stato creare i file JSON “map.json” per la *MapView* ed il file “manifest.json”. Si è deciso di usare come formato il JSON a discapito di XML per avere meno overhead in quanto necessita di una struttura più snella e quindi un file meno pesante. Il metodo più rapido per convertire un testo JSON in un oggetto .NET è utilizzare la classe *JsonSerializer*. Essa converte gli oggetti .NET nel loro equivalente JSON e viceversa mappando i nomi delle proprietà degli oggetti .NET nei nomi delle proprietà JSON e ne copia i valori.

```
public string JSONString(PluginDescriptor plugin)
{
    List<MapViewClassDescriptor> maps = new List<MapViewClassDescriptor>();
    foreach (var c in plugin.Classes)
    {
        foreach (var m in c.MapView.MapViewItems)
        {
            MapViewClassDescriptor mapViewClass = new MapViewClassDescriptor(c.Name, m);
            maps.Add(mapViewClass);
        }
    }

    var jsonSerializerSettings = new JsonSerializerSettings()
    {
        TypeNameHandling = TypeNameHandling.All
    };
    var json = JsonConvert.SerializeObject(maps, jsonSerializerSettings);

    return json;
}
```

Figura 18 – Metodo JSONString

Il metodo appena citato è stato utilizzato per la creazione di entrambi i file JSON cambiando semplicemente il “value” nel metodo *SerializeObject*. Nella figura mostrata sopra si evince che il metodo è stato usato per la creazione del file “map.json”. Per il file “manifest.json” è stato necessario modificare il “value” da “maps” ad un oggetto *LoaderManifest* creato appositamente in precedenza contenente tutte le proprietà richieste (*Name*, *Version*, *DLLs*, *Extension* e *TemplateExtension*).

Il file “map.json” è fondamentale per poter essere in grado di esprimere per ogni classe (AND, OR, NOT, XOR, NOR, NAND, NOR) un simbolo, secondo le proprietà da questo esposte, cioè dobbiamo essere in grado di associare un simbolo ad un determinato stato di un oggetto.

```
{
  "$type": "System.Collections.Generic.List`1[[SymbolEditor.dto.MapViewClassDescriptor, SymbolEditor]], System.Private.CoreLib",
  "$values": [
    {
      "$type": "SymbolEditor.dto.MapViewClassDescriptor, SymbolEditor",
      "FileName": "logicgate.svg",
      "Description": "Logic gate con due input",
      "ExportIntoDocumentation": true,
      "SampleData": {
        "$type": "System.Collections.Generic.Dictionary`2[[System.String, System.Private.CoreLib],[System.String, System.Private.CoreLib]], System.Private.CoreLib",
      },
      "Properties": {
        "$type": "System.Collections.Generic.Dictionary`2[[System.String, System.Private.CoreLib],[System.String, System.Private.CoreLib]], System.Private.CoreLib",
        "ClassName": "LogicGate",
        "NumberOfInput": "2",
        "NumberOfOutput": "1"
      }
    }
  ],
}
```

Figura 19 – Esempio parte iniziale file map.json

Il file “manifest.json” è colui che esprime il contenuto dell’intera applicazione, cioè il suo manifesto nel senso letterale del termine, considerando che dentro sarà presente tutto ciò che serve a “CADdy” per interpretare la GA.

```
{
  "$type": "SymbolEditor.dto.LoaderManifest, SymbolEditor",
  "Name": "LogicGate",
  "Version": "1.0.0",
  "DLLs": {
    "$type": "System.Collections.Generic.List`1[[System.String, System.Private.CoreLib]], System.Private.CoreLib",
    "$values": [
      "LogicGate.dll"
    ]
  },
  "Extension": {
    "$type": "System.Collections.Generic.List`1[[System.ValueTuple`2[[System.String, System.Private.CoreLib],[System.String, System.Private.CoreLib]], System.Private.CoreLib]], System.Private.CoreLib",
    "$values": [
    ]
  },
  "TemplateExtension": {
    "$type": "System.Collections.Generic.List`1[[System.ValueTuple`2[[System.String, System.Private.CoreLib],[System.String, System.Private.CoreLib]], System.Private.CoreLib]], System.Private.CoreLib",
    "$values": [
    ]
  }
}
```

Figura 20 – Esempio file manifest.json

Si è deciso di usare il formato JSON (JavaScript Object Notation) per i seguenti motivi:

- È facile da leggere e da scrivere.
- È un formato di interscambio leggero basato su testo in quanto semplifica i documenti complessi fino ai componenti che sono stati identificati come significativi.
- È indipendente dal linguaggio di programmazione scelto, quindi perfetto per fornire un elevato livello di interoperabilità.
- Ha un formato dei dati facile da analizzare che non richiede codice aggiuntivo per l'analisi.

L'ultimo passo prima della creazione dello zip con estensione GA è stato fare il matching tra i path dei file SVG relativi al file XML per ricavare il path assoluto dei file SVG. Per ottenere questo risultato è stato necessario l'utilizzo del metodo *Path.Combine*, già citato in questo elaborato. Avendo il path assoluto di tutti i componenti SVG, il passo successivo da compiere è stato quello di copiare i rispettivi file nella cartella contenente gli altri file.

Dopo aver ottenuto tutti i file necessari per lo zip GA, è stato implementato un metodo per inserirli tutti in una cartella ed averli così pronti per essere importati in Caddy, estratti, riconvertiti in oggetti riconosciuti dal software medesimo ed infine essere utilizzati dall'utente.

5.2 Log4net

Quando si lavora sulle applicazioni, è possibile che si desideri spesso registrare i dati dell'applicazione che possono includere la sequenza di eventi nell'applicazione, le azioni dell'utente o persino gli errori (quando si verificano). Esistono molti framework di registrazione che si possono utilizzare, ma log4net è di gran lunga uno dei framework di registrazione più popolari da utilizzare con le applicazioni create o sviluppate in .NET (l'intero progetto è stato sviluppato in Visual Studio 2022). È una libreria open source, facile da usare, affidabile, veloce, popolare ed estensibile che può essere utilizzata per registrare i dati dell'applicazione su diverse destinazioni di log in .NET. [10]

Log4net fornisce un semplice meccanismo per la registrazione delle informazioni in una varietà di fonti. Le informazioni vengono registrate tramite uno o più logger. Questi logger forniscono 5 livelli di logging:

- Debug
- Info
- Warn
- Error
- Fatal

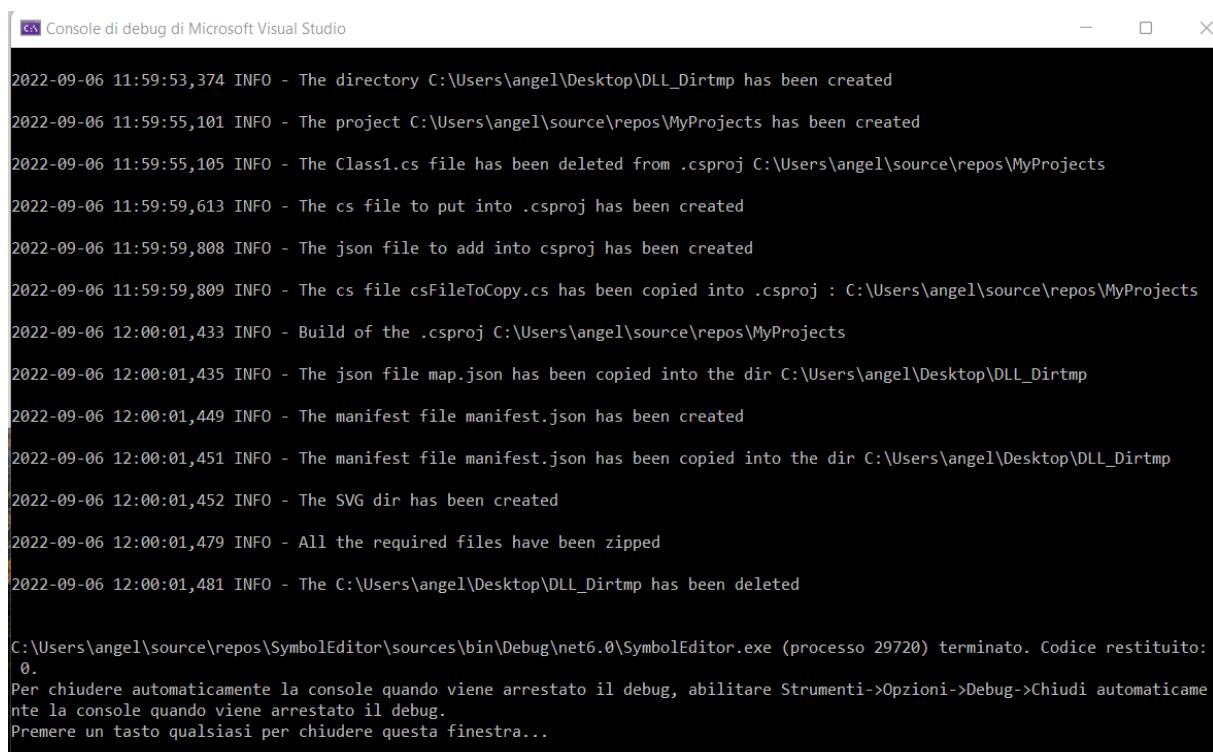
L'obiettivo è che la quantità di log eseguita da ciascun livello diminuisca scendendo in questo elenco (vogliamo più informazioni di debug o info che fatal error). È possibile specificare il livello di log su un particolare logger, quindi durante lo sviluppo è possibile emettere tutti e 5 i livelli di logging. Le informazioni di logging vanno a finire in quello che viene chiamato Appender. Un appender è fondamentalmente una destinazione in cui andranno le informazioni di log (ad esempio *ConsoleAppender* e *FileAppender*). Esistono molti altri appender per consentire ai dati di essere registrati su database, e-mail, trasmissioni di rete ecc. Il programmatore non è limitato nell'utilizzo di un solo appender, si possono avere molteplici appender configurati per l'uso che si desidera. La configurazione dell'appender viene eseguita al di fuori del codice nei file XML, quindi, è semplice modificare la configurazione di logging.

Le informazioni verranno registrate tramite diversi layout che possono essere usati per ogni appender. Questi layout specificano se i log vengono prodotti come semplici dati testuali o come file XML, o se hanno timestamp, ecc. [11]

Avere dei log file è un modo per avere la cronologia di tutto ciò che si verifica nell'utilizzo del nostro software. Ciò contribuirà a tenere traccia dell'utilizzo delle risorse del sistema ed anticipare e correggere eventuali guasti prima che si verifichino effettivamente.

Nel progetto i log sono stati usati principalmente per tenere traccia delle varie operazioni svolte, come ad esempio la creazione del progetto libreria, l'eliminazione del file Class1.cs presente in ogni nuovo progetto VS, la creazione dei file con estensione json, l'aggiunta e l'eliminazione di file, la creazione della cartella compressa contenente tutti i file utili per la creazione del nuovo tool nel CAD.

Nella figura sottostante si può osservare un esempio pratico di utilizzo dei log nel progetto e di quanto sia chiara la tracciabilità delle singole operazioni.



```
Console di debug di Microsoft Visual Studio

2022-09-06 11:59:53,374 INFO - The directory C:\Users\angel\Desktop\DLL_Dirtmp has been created
2022-09-06 11:59:55,101 INFO - The project C:\Users\angel\source\repos\MyProjects has been created
2022-09-06 11:59:55,105 INFO - The Class1.cs file has been deleted from .csproj C:\Users\angel\source\repos\MyProjects
2022-09-06 11:59:59,613 INFO - The cs file to put into .csproj has been created
2022-09-06 11:59:59,808 INFO - The json file to add into csproj has been created
2022-09-06 11:59:59,809 INFO - The cs file csFileToCopy.cs has been copied into .csproj : C:\Users\angel\source\repos\MyProjects
2022-09-06 12:00:01,433 INFO - Build of the .csproj C:\Users\angel\source\repos\MyProjects
2022-09-06 12:00:01,435 INFO - The json file map.json has been copied into the dir C:\Users\angel\Desktop\DLL_Dirtmp
2022-09-06 12:00:01,449 INFO - The manifest file manifest.json has been created
2022-09-06 12:00:01,451 INFO - The manifest file manifest.json has been copied into the dir C:\Users\angel\Desktop\DLL_Dirtmp
2022-09-06 12:00:01,452 INFO - The SVG dir has been created
2022-09-06 12:00:01,479 INFO - All the required files have been zipped
2022-09-06 12:00:01,481 INFO - The C:\Users\angel\Desktop\DLL_Dirtmp has been deleted

C:\Users\angel\source\repos\SymbolEditor\source\bin\Debug\net6.0\SymbolEditor.exe (processo 29720) terminato. Codice restituito:
0.
Per chiudere automaticamente la console quando viene arrestato il debug, abilitare Strumenti->Opzioni->Debug->Chiudi automaticame
nte la console quando viene arrestato il debug.
Premere un tasto qualsiasi per chiudere questa finestra...
```

Figura 21 – Utilizzo dei log nel progetto

Capitolo 6 - GUI e Testing

6.1 Introduzione GUI SymbolEditor

Per agevolare ulteriormente l'interazione con l'utente si è deciso di implementare una GUI (Graphical User Interface) per aprire i file XML, modificarli in maniera più semplice ed intuitiva e per crearne di nuovi.

L'interazione dell'utente con gli elementi grafici presenti nell'interfaccia grafica ha sostituito la comunicazione uomo-macchina basata sulle linee di comando, interazione tipica dei primi sistemi operativi. L'interfaccia grafica ha quindi fornito agli utenti un sistema relativamente intuitivo che ha reso l'utilizzo del computer più facile da imparare, e più piacevole e naturale rispetto alle difficili linee di comando, diffondendo l'uso dei computer anche tra i non tecnici.

Una struttura GUI permette di creare un'applicazione usando tutta una serie di elementi GUI, quali *label*, *button*, *textbox*, *checkbox*, *comboBox*, *listView* ed innumerevoli altri. Senza una struttura GUI sarebbe necessario sia disegnare manualmente questi elementi che gestire tutti gli scenari di interazione con l'utente come, ad esempio, gli input testuali oppure del mouse. Questo comporterebbe molto lavoro, ecco perché è particolarmente diffusa la creazione di una struttura GUI a cui delegare tutto il lavoro di base.

La lettura del template XML citato in questo elaborato risulta, soprattutto senza la presenza di alcun tipo di documentazione, di gran lunga di più difficile comprensione rispetto ad una visione grafica.

6.2 WPF e XAML

L'interfaccia grafica è stata realizzata completamente in WPF (Windows Presentation Foundation). Windows Presentation Foundation è un framework per lo sviluppo dell'interfaccia utente delle applicazioni in ambiente Windows. La piattaforma di sviluppo WPF supporta un'ampia serie di funzionalità di sviluppo di applicazioni, inclusi un modello applicativo, risorse, elementi grafici, data binding, controlli, layout, elementi grafici, documenti e sicurezza.

Le applicazioni sviluppate in WPF sono composte da 2 thread: il thread per gestire la UI e l'altro thread detto render thread che in maniera nascosta gestisce le funzioni di rendering e repainting. Perciò rendering e repainting sono gestiti da WPF stesso, senza intervento dello sviluppatore.

Il thread per la UI ospita il *Dispatcher* (attraverso un'istanza dell'oggetto *DispatcherObject*), il quale mantiene una coda di operazioni che bisogna eseguire sulla UI, ordinate per priorità. Gli eventi della UI, ad esempio il cambiamento di una proprietà che riguarda il layout, ed eventi causati dall'interazione dell'utente sono accodati nel *Dispatcher* che avrà il compito di invocare i gestori degli eventi. È raccomandabile che i gestori degli eventi aggiornino solo le proprietà per riflettere il nuovo contenuto come risposta; il nuovo contenuto sarà generato o recuperato dal render thread. Il render thread, inoltre, salva in una cache l'albero visuale, così dovranno essere comunicati solo i cambiamenti all'albero, e perciò soltanto i pixel cambiati risulteranno aggiornati. [12]

WPF usa il linguaggio XAML (eXtensible Application Markup Language) per offrire un modello dichiarativo per la programmazione di applicazioni. XAML è un linguaggio di markup dichiarativo basato su XML. Come tutti i linguaggi di markup, anche XAML usa i tag per definire gli oggetti, mentre gli attributi di un oggetto sono definiti all'interno dei tag. Si possono creare elementi della GUI visibili nel markup XAML dichiarativo, quindi separare la definizione dell'interfaccia utente dalla logica di runtime utilizzando dei file code-behind uniti al markup tramite definizioni di classe parziali. XAML consente un workflow in cui parti distinte possono operare nell'interfaccia utente e nella logica di un'applicazione, usando strumenti potenzialmente diversi. [13]

I file “.xaml” permettono la descrizione dell'interfaccia tramite tag e “collegamenti”, detti *Binding*, ad oggetti e azioni contenuti nel *ViewModel* associatogli, che ne diviene il *DataContext*.

6.3 Sezioni GUI

La GUI si presenta vuota all'inizio. In questo stato permette all'utente, comunque, di inserire tutti i dati necessari e tramite il bottone “Select File” di fianco alla voce “Save data” permette di specificare il path che si vuole sovrascrivere oppure di creare dinamicamente un nuovo file XML. Si è deciso di sfruttare un oggetto *TabControl* per spostarsi da una sezione all'altra. In WPF il *TabControl* consente di dividere l'interfaccia in aree differenti, ciascuna accessibile cliccando sull'intestazione del tab, solitamente posizionato nella parte superiore del controllo. Il *TabControl* è utile per ridurre al minimo l'utilizzo dello spazio dello schermo consentendo ad un'applicazione di esporre una grande quantità di dati. Un oggetto *TabControl* è costituito da

più oggetti *TabItem* che condividono lo stesso spazio sullo schermo. È possibile vedere a schermo un solo un oggetto *TabItem* alla volta. Quando un utente seleziona la sezione di un oggetto *TabItem*, il contenuto di un *TabItem* diventa visibile e il contenuto degli altri oggetti *TabItem* rimangono nascosti. Nell'intero sviluppo della parte grafica non si è usato *Windows.Forms* così da poter trasferire in maniera agevole la GUI anche su piattaforma Android.

Si è deciso di dividere il *TabControl* in quattro sezioni differenti consistenti in “General”, “Input”, “Type” e “View”.

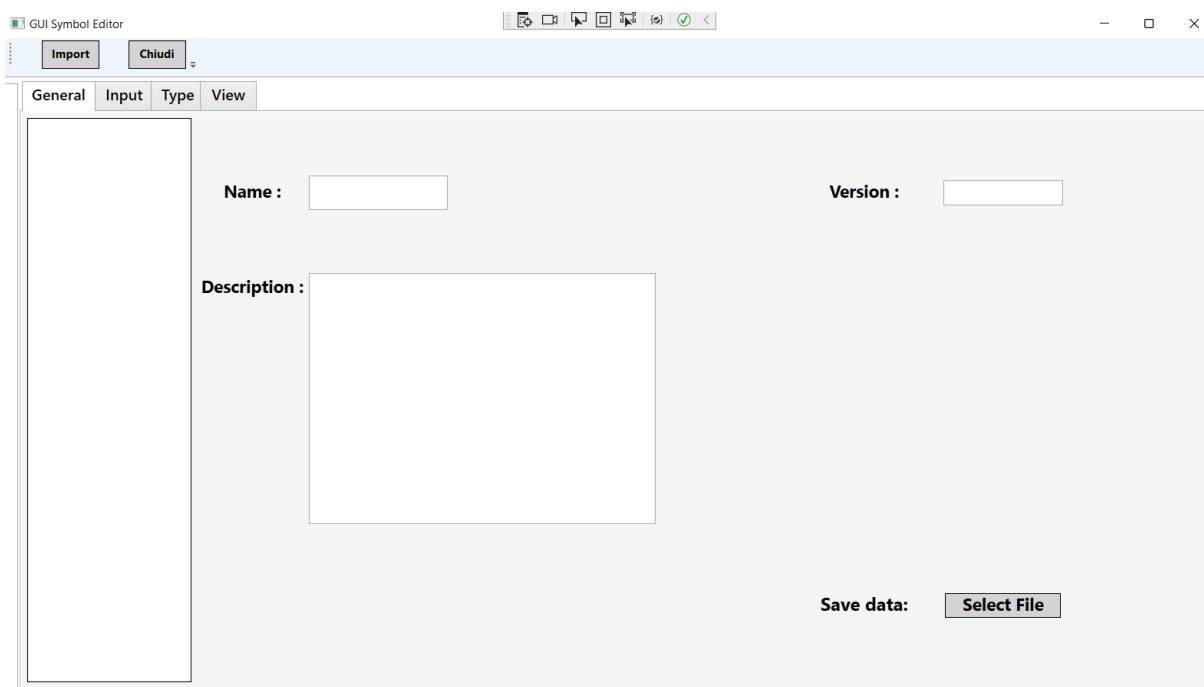


Figura 22 – Interfaccia iniziale GUI

In alto è presente un componente *ToolBarTray*. Esso rappresenta il contenitore che gestisce il layout di un oggetto *ToolBar* al cui interno sono contenuti due bottoni: “Import” e “Chiudi”. Il primo consente di importare un file XML tramite apertura di un *File Dialog*. Per non permettere all'utente di importare qualsiasi tipo di file è stato necessario fare un controllo sul file selezionato, e nel caso di selezione di un file non conforme alle specifiche, la conseguente visualizzazione a schermo di un *MessageBox* che informi l'utente dell'errata selezione del file, con conseguente possibilità di selezionare un nuovo file.

Nella parte sottostante la *ToolBarTray* è presente l'oggetto *TabControl* con i suoi quattro oggetti *TabItem* già accennati in precedenza.

Nella parte sinistra della schermata è presente un componente *ListView* che si popolerà con il nome delle classi presenti nel file XML non appena quest'ultimo sarà importato. Questa non sarà una semplice lista bensì una *ObservableCollection*, ovvero una raccolta di dati dinamica che fornisce notifiche quando gli elementi vengono aggiunti, rimossi o quando l'intero elenco viene aggiornato. Per aggiornare automaticamente la schermata, la classe *ViewModel* ha dovuto implementare l'interfaccia *INotifyPropertyChanged*. Quest'ultima notifica al client che è stato modificato un valore di proprietà.

Nella parte destra troviamo diversi label con a fianco i *TextBox* che si riempiranno non appena verrà importato un file XML. Questi ultimi sono modificabili così da permettere eventualmente all'utente di cambiarne i valori.

In basso a destra è presente il bottone "Save data" già accennato nella pagina precedente.

Tutti i bottoni presenti nella GUI hanno implementato l'interfaccia *ICommand*. I "comandi" forniscono, nell'architettura MVVM (Model-View-ViewModel) un meccanismo per la View di aggiornare il Model. L'interfaccia *ICommand* presenta due metodi ed un evento.

- Il metodo *CanExecute* prende un oggetto come argomento e restituisce un bool. Se restituisce "true", è possibile eseguire il comando associato. Se restituisce "false", il comando associato non può essere eseguito. Per conoscere il valore di *CanExecute* si guarda all'evento *CanExecuteChanged*, che può variare in base al parametro passato.
- l'evento *CanExecuteChanged* viene utilizzato per notificare ai controlli dell'interfaccia utente associati al "comando" se esso può essere eseguito. In base alla notifica di questo evento, i controlli dell'interfaccia utente ne modificano lo stato in abilitato o disabilitato.
- Il metodo *CanExecute* è colui che esegue il lavoro effettivo previsto per il "comando". Questo metodo viene eseguito solo se il metodo *CanExecute* restituisce "true". Prende un oggetto come argomento e generalmente viene passato un delegato in questo metodo. Il delegato contiene un riferimento al metodo che dovrebbe essere eseguito quando viene attivato il comando. [14]

In questo progetto si è deciso di usare *RelayCommand*, un'implementazione di *ICommand* che permette di esporre un metodo alla vista. Questa implementazione permette di specificare come argomento un metodo che implementi l'azione da svolgere in seguito all'avvenuto click sul bottone. Il delegato che viene usato nel metodo *CanExecute* è un evento di tipo *PropertyChangedEventHandler*. Esso rappresenta il metodo tramite il quale verrà gestito l'evento *PropertyChanged* generato quando verrà modificata una proprietà su un componente.

Nella seguente figura si può osservare un esempio d'uso del *RelayCommand*, in particolare dell'import di un nuovo file XML.

```
private ICommand _CMDButton_Click_Import;
0 riferimenti
public ICommand CMDButton_Click_Import
{
    get
    {
        return _CMDButton_Click_Import ?? (_CMDButton_Click_Import = new RelayCommand(
            x => {
                Button_Click_Import();
            }));
    }
}
```

Figura 23 – Esempio d'uso *RelayCommand*

6.3.1 Sezione General

Dopo aver importato un file XML verrà creato un oggetto *PluginDescriptor* e di conseguenza tutti i sotto oggetti in esso contenuti, nonché la lista dei DLL, la lista dei NuGet Packages e la lista degli SVG.

Verranno inoltre visualizzati a schermo tutti i campi, dando comunque la possibilità all'utente di modificarli. Tutti i *TextBox*, durante la modifica dei campi da parte dell'utente, forzeranno l'*ICommand* ad aggiornarsi automaticamente tramite il metodo *CommandManager.InvalidateRequerySuggested()*. Esso procurerà il trigger del delegato *PropertyChangedEventHandler* del solo campo modificato, tramite la specifica di "nameof" del campo da aggiornare.

```
private string _PluginNameText;
5 riferimenti
public string PluginNameText
{
    get { return _PluginNameText; }
    set
    {
        if (_PluginNameText != value)
        {
            _PluginNameText = value;
            // Force the ICommand to update
            CommandManager.InvalidateRequerySuggested();

            // Raise property changed!
            var handler = this.PropertyChanged;
            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(nameof(PluginNameText)));
            }
        }
    }
}
```

Figura 24 – Esempio aggiornamento campo

Si otterrà così un oggetto *PluginDescriptor* “temporaneo” che sarà aggiornato costantemente in base ai cambiamenti che l’utente apporterà all’oggetto tramite l’interfaccia grafica. Una volta completati gli aggiornamenti sul file, l’utente potrà salvare quest’ultimo premendo il bottone “Save data”. Così facendo si aprirà un elemento di controllo *SaveFileDialog* che permetterà di selezionare il percorso dove salvare il file. Per Serializzare nuovamente l’oggetto *PluginDescriptor* in un file XML sarà necessario avvalersi della classe *XmlSerializer*. Questa volta però la serializzazione risulta più facile ed immediata, essendo già stata implementata nella fase di Model. Nella figura riportata di seguito si può osservare la fase di serializzazione da oggetto a file XML.

```
System.Xml.Serialization.XmlSerializer x = new System.Xml.Serialization.XmlSerializer(typeof(PluginDescriptor));

if (saveFileDialog.ShowDialog() == true)
{
    // File.WriteAllText(saveFileDialog.FileName, plugin);
    using var myXML = new FileStream(saveFileDialog.FileName, FileMode.Open);
    {
        x.Serialize(myXML, plugin);
    }
}
```

Figura 25 – Esempio serializzazione in GUI

Di seguito riportata la figura che mostra la sezione General dopo l’import di un file XML di esempio.

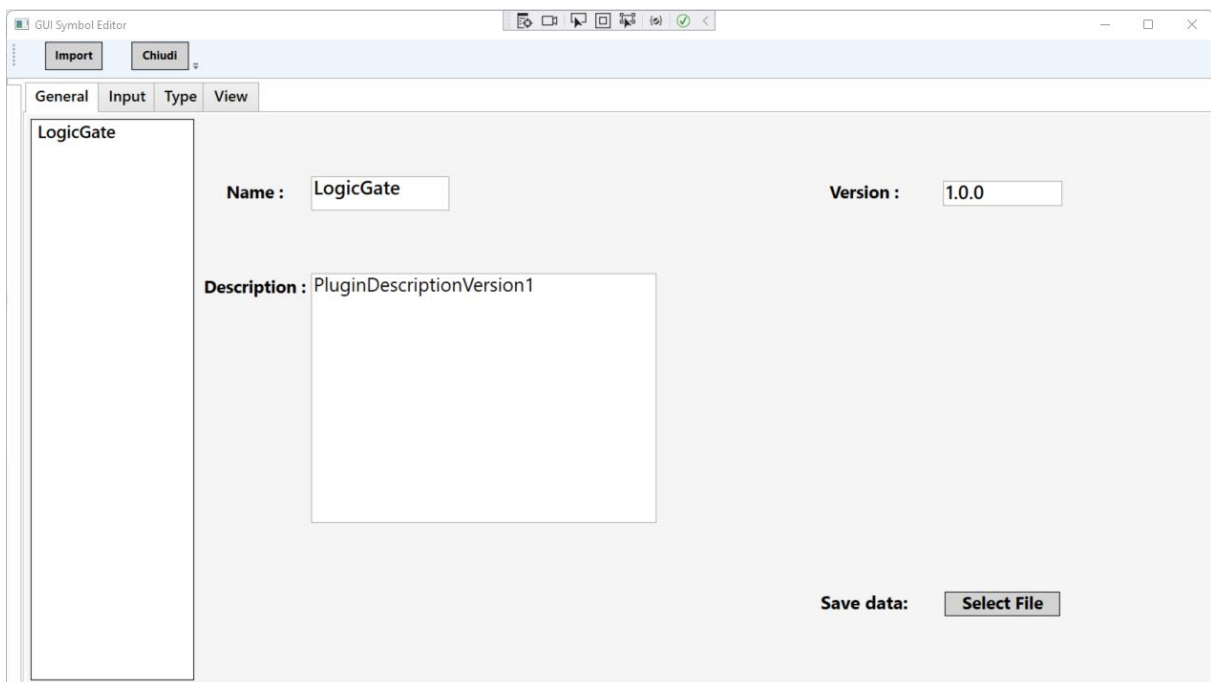


Figura 26 – Sezione “General” GUI

6.3.2 Sezione Input

Spostando l'attenzione sul *TabItem* denominato "Input" troviamo la sezione inerente ai file DLL e ai NuGet Packages. Premendo il bottone "Select Files" di fianco al label denominato "DLL Files" è possibile importare file con estensione ".dll". È presente una verifica che i file selezionati siano effettivamente dei file DLL. È inoltre possibile selezionare più file in quanto è stata impostata l'opzione *Multiselect* su "true". Completata la fase di selezione dei file DLL, essi sono visibili nella *ListView* presente sulla parte destra della schermata. Quest'ultima può essere modificata dall'utente selezionando il nome del file DLL che si vuole eliminare dalla lista e premendo il bottone rosso. Dopo l'avvenuta eliminazione del DLL, la *ObservableCollection* di DLL verrà aggiornata. Nella parte sottostante è possibile inserire, nell'apposita *TextBox* di fianco al label denominato "Nuget Packages", una stringa contenente il nome di un NuGet package da importare nel progetto e, tramite l'apposito bottone "+", aggiungerlo in una *ObservableCollection* per poter sempre tener traccia di tutti i NuGet packages già impostati. Non è possibile aggiungere più NuGet packages aventi lo stesso nome in quanto è presente un check che controlli, prima di aggiungere un nuovo package, se esso sia già presente nella lista o meno. Anche in questo caso, l'utente potrà osservare ogni cambiamento apportato in una *ListView* nella parte inferiore dello schermo. È inoltre previsto un bottone per rimuovere dalla lista i NuGet packages non necessari.

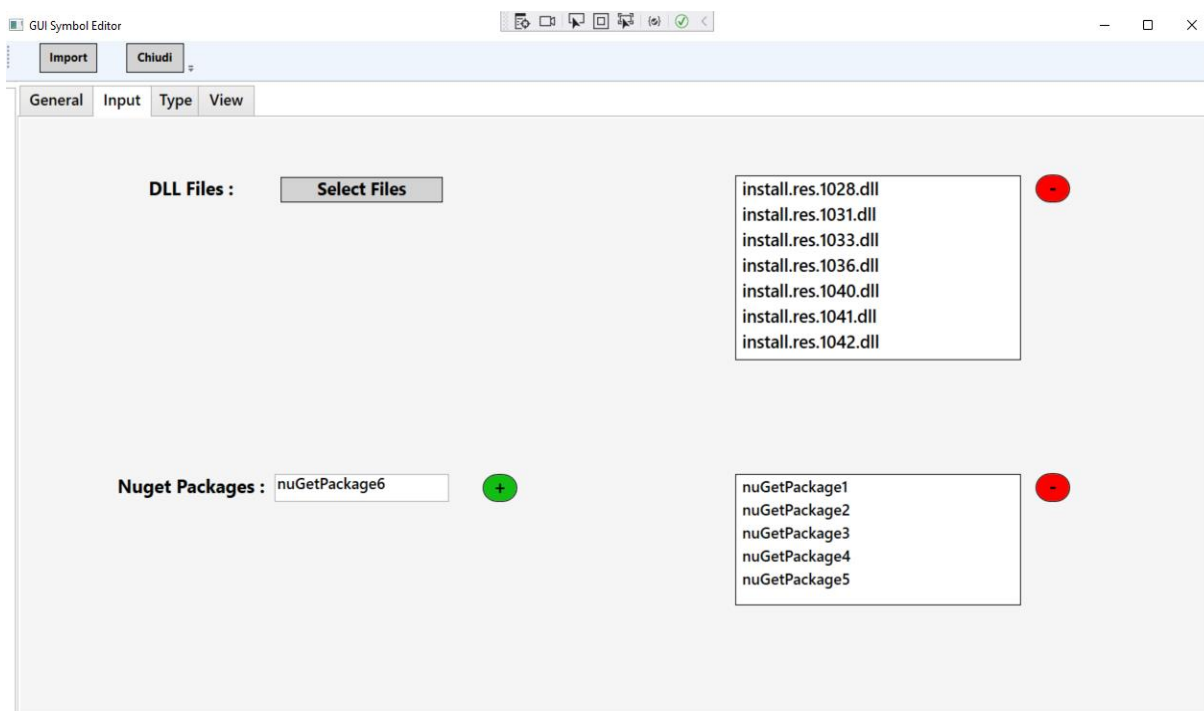


Figura 27 – Sezione "Input" GUI

6.3.3 Sezione Type

Proseguendo l'analisi dei *TabItem* presenti nella GUI, la sezione denominata "Type" presenta tutti i tipi presenti nel Plugin (i *ClassDescriptor* già citati molteplici volte all'intero dell'elaborato), con le loro proprietà e la classe che estendono.

Nella parte sinistra troviamo una *ListView* contenente tutti i nomi dei tipi. Selezionando uno di essi si riempirà un *TextBox* contenente il nome del tipo, una lista che mostra quali classi estende e una seconda *ListView*, presente al centro dello schermo, dove è possibile osservare tutte le proprietà. Queste possono essere rimosse attraverso il bottone "- Remove" oppure aggiunte di nuove attraverso il bottone "+ Add". Premendo quest'ultimo bottone si triggerà l'evento "click" che a sua volta farà apparire un nuovo pannello di tipo *UserControl* nella parte destra dello schermo. Lo scopo di uno *UserControl* è raggruppare un insieme di controlli in un componente riutilizzabile. All'interno sono presenti tutti i campi per l'inserimento di una nuova proprietà (nome, tipo e descrizione). Attraverso il bottone "+ Add New", i dati presenti nelle tre *TextBox* vengono salvati e aggiunti alla lista delle proprietà della classe. Non è possibile aggiungere due proprietà identiche in quanto è presente un confronto dei dati che si vogliono inserire con la lista delle proprietà del tipo selezionato. In caso la proprietà che si vuole aggiungere sia già presente, verrà mostrato a schermo un *MessageBox* che informi l'utente di questo evento, dandogli comunque la possibilità di inserire altre proprietà.

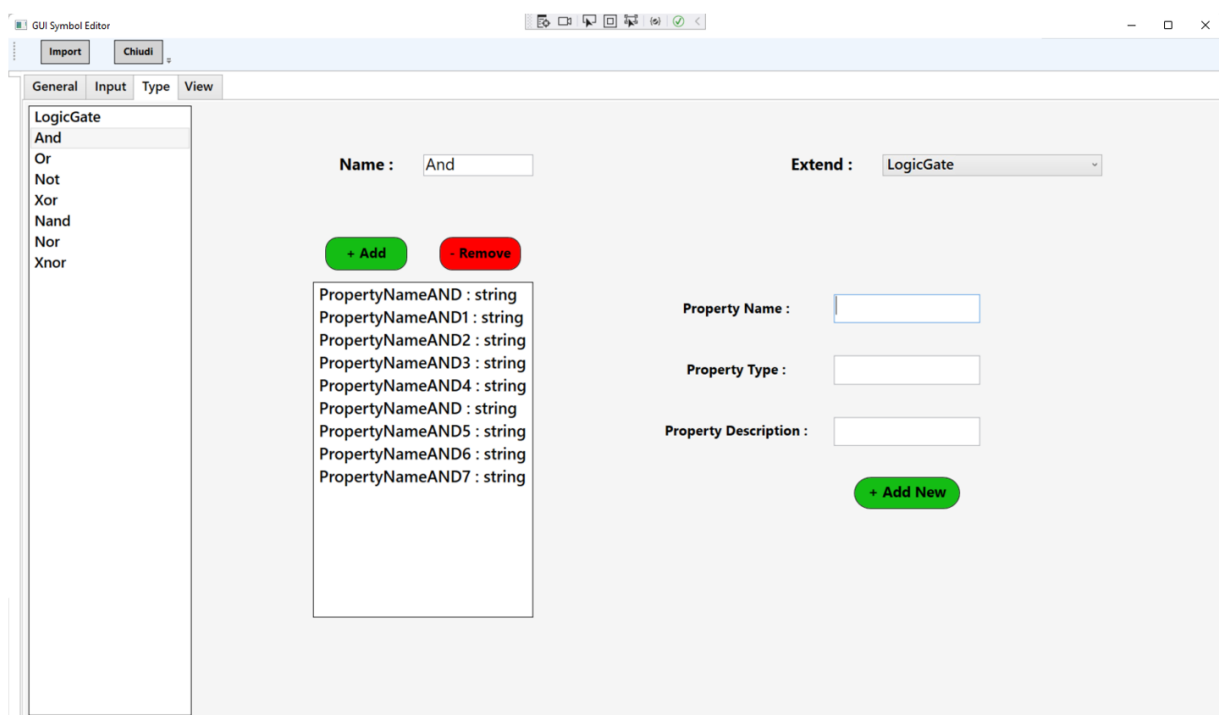


Figura 28 – Sezione "Type" GUI

6.3.4 Sezione View

L'ultimo *TabItem* presente nel *TabControl* è rappresentato dalla sezione View. Come il nome fa ben intuire, in esso saranno presenti tutte le componenti del Plugin facenti parte della parte visiva.

Nella parte sinistra troviamo una *ListView* contenente tutti i nomi dei tipi, esattamente come nella sezione Type. Al centro è presente un *TextBox* dove sarà visualizzato il codice di markup che caratterizza il tipo selezionato. In presenza di codice SVG valido sarà visualizzato, tramite un *UserControl*, l'SVG corrispondente al codice. È stato necessario implementare un componente *UserControl* (chiamato *SVGControl*) che riuscisse a convertire la stringa presente nel *TextBox* sottostante il label denominato "SVG" nel corrispettivo grafico SVG. Modificando il testo scritto nel *TextBox* è possibile vedere i cambiamenti grafici sulla visualizzazione dell'oggetto di grafica vettoriale SVG.

```
public partial class SVGControl : System.Windows.Controls.UserControl
{
    private static readonly ILog _Log = LogManager.GetLogger(typeof(SVGControl));

    1 riferimento
    public string SvgData
    {
        get { return (string)GetValue(SvgDataProperty); }
        set { SetValue(SvgDataProperty, value); }
    }
    public static readonly DependencyProperty SvgDataProperty = DependencyProperty.Register("SvgData", typeof(string), typeof(SVGControl), new PropertyMetadata(string.Empty,
        (depObj, args)=>
        {
            try
            {
                if (depObj is SVGControl This && args.NewValue is string str && str.Length > 0)
                {
                    var bytes = Encoding.ASCII.GetBytes(str).ToList();
                    using var stream = new MemoryStream(bytes.ToArray());
                    This.canvas.StreamSource = stream;
                }
            }
            catch (Exception ex)
            {
                _Log.Error(ex.Message);
            }
        }
    ));
    0 riferimenti
    public SVGControl()
    {
        InitializeComponent();
    }
}
```

Figura 29 – Implementazione classe SVGControl

Sulla parte destra dello schermo saranno visualizzate, in una *ListView*, tutte le proprietà del tipo selezionato in sola lettura con, in aggiunta, le proprietà create dall'utente nella sezione Type.

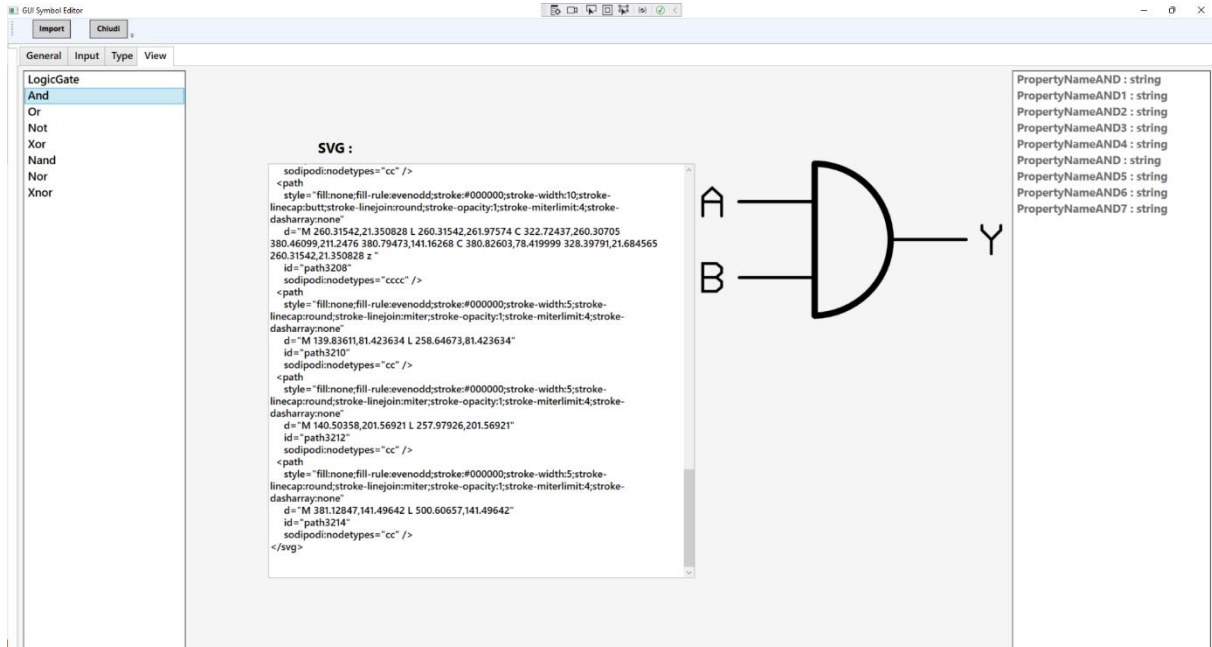


Figura 30 – Sezione “View” GUI

6.4 Testing

Per testare la reale efficienza e validità del SymbolEditor è stato preso un file XML, lo si è modificato e si è sovrascritto il contenuto sullo stesso file. Successivamente, si è preso lo stesso file, analizzato e salvato su un nuovo file XML. Si è sfruttato il software “Beyond Compare 4” per confrontare i due file. Beyond Compare è un software multiplatforma specializzato nel confronto dei dati. Oltre a confrontare i file, il programma è in grado di eseguire confronti affiancati di directory, directory FTP e SFTP, directory Dropbox, directory Amazon S3 e archivi.

I due file XML sopracitati sono stati confrontati sia come “Text Compare” che come “Hex Compare”. In entrambi i casi i file risultavano identici. Possiamo quindi capire che la fase di serializzazione e deserializzazione sviluppata sia nella parte di Model che nella parte di View-Model funziona correttamente e omogeneamente.

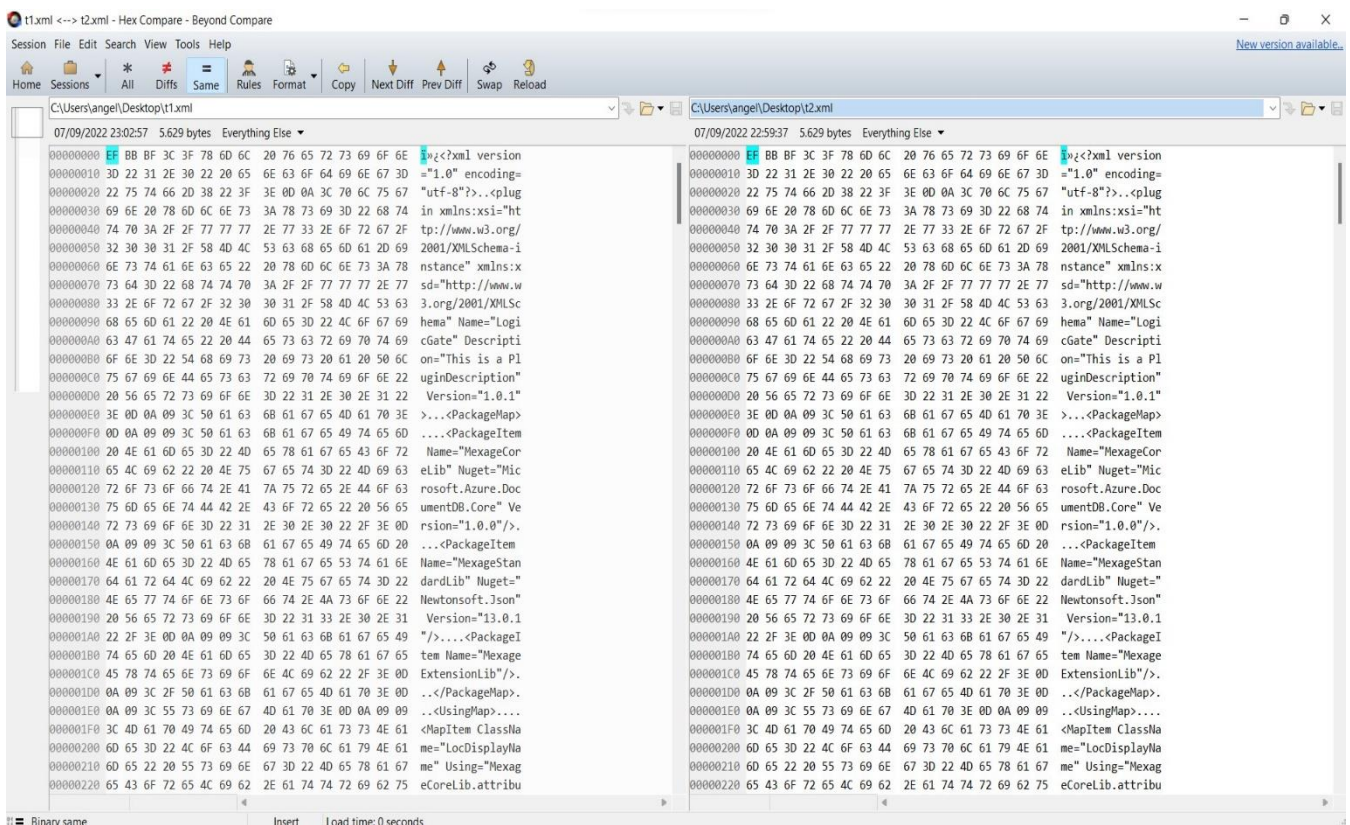


Figura 31 – Confronto HEX di due file XML

Capitolo 7 - Conclusioni e Sviluppi Futuri

Ricapitolando, questo elaborato mira ad analizzare lo sviluppo di un editor grafico per la scrittura di codice. Al posto di un IDE classico in cui scrivere del testo che verrà riconosciuto come codice, è stato ideato un editor grafico che permetta di definire degli oggetti che poi verranno tradotti in codice. È stata sviluppata sia la parte di back end che di front end. Questo editor grafico è a tutti gli effetti un plugin per CAD 2D generico che consente la creazione e l'inserimento di nuovi oggetti dentro il CAD stesso. Lo scopo finale dell'elaborato è stato quello di creare nuovi oggetti ed utilizzarli nel software "CADdy" sviluppato internamente da MeXage. Il punto di partenza è stato la creazione della struttura del template XML. Si è poi passati ad una fase di Deserializzazione del file XML in classi con conseguente Serializzazione, avvalendosi del Visitor Design Pattern, in un solo file CS contenente tutti gli oggetti dichiarati nel file XML. La build del progetto in DLL e la creazione di tutti i file necessari all'importazione dell'oggetto in CADdy tramite un file zip con estensione GA ha permesso di usufruire dell'oggetto direttamente in CADdy, avendo addirittura un'icona grafica rappresentante l'oggetto realizzato. Finita la parte di back end, è stata sviluppata una interfaccia grafica in WPF, per permettere all'utente una più immediata comprensione del file XML, la sua modifica e sovrascrittura senza perdita di informazioni. L'utente è in grado di aggiungere nuove proprietà al tipo e rimuovere le proprietà che ritiene non necessarie. Grazie al componente *SVGControl* è possibile visualizzare in tempo reale il codice di markup e la sua versione visiva. Modificando il *TextBox* sarà modificato anche questo componente con conseguente modifica della sua rappresentazione.

È stata necessaria una attenta fase di sviluppo degli algoritmi che ha richiesto numerosi momenti di studio ed analisi teorica. Inoltre, è stata necessaria una profonda comprensione del CAD per riuscire a sviluppare software che comunichi in maniera omogenea con quest'ultimo. Possiamo affermare che il plugin sviluppato abbia raggiunto gli obiettivi preposti. La forte genericità del progetto permette di non essere necessariamente vincolati nell'uso di "CADdy", bensì di poterlo adoperare in qualsiasi tipo di CAD generico. Questo ha permesso di lasciare aperta una finestra verso il futuro, ad esempio dando la possibilità di avere un'applicazione mobile.

Bibliografia e Sitografia

- [0] MeXage. *MeXage – Engineering your ideas* - <https://www.mexage.net/> - Visitato agosto 2022
- [1] Medium - *Meta-Programming : Code that writes Code*
<https://anshul-vyas380.medium.com/meta-programming-code-that-writes-code-fde1d84da4b#:~:text=Meta-Programming%20is%20a%20programming,even%20modify%20itself%20while%20running> - Visitato agosto 2022
- [2] Quora - *What is metaprogramming, and why is it important to learn it as a programmer?*
<https://www.quora.com/What-is-metaprogramming-and-why-is-it-important-to-learn-it-as-a-programmer> - Visitato agosto 2022
- [3] ReadItalian - *Metaprogrammazione* - <https://readitaliano.com/wiki/it/Metaprogramming> - Visitato agosto 2022
- [4] Microsoft - *ProcessStartInfo Classe*
<https://docs.microsoft.com/it-it/dotnet/api/system.diagnostics.processstartinfo?view=net-6.0> - Visitato settembre 2022
- [5] Microsoft - *Informazioni di base su XML*
<https://support.microsoft.com/it-it/office/informazioni-di-base-su-xml-a87d234d-4c2e-4409-9cbc-45e4eb857d44#:~:text=Recupero%20più%20facile%20delle%20informazioni,almeno%20in%20parte%20le%20informazioni> - Visitato settembre 2022
- [6] Microsoft - *Come deserializzare un oggetto usando XmlSerializer*
<https://docs.microsoft.com/it-it/dotnet/standard/serialization/how-to-deserialize-an-object> - Visitato settembre 2022
- [7] Microsoft – *Serializzazione (C#)*
<https://docs.microsoft.com/it-it/dotnet/csharp/programming-guide/concepts/serialization/> - Visitato settembre 2022
- [8] SourceMaking – *Visitor Design Pattern* - https://sourcemaking.com/design_patterns/visitor - Visitato settembre 2022
- [9] ItalianCoders – *Visitor Pattern* - <https://italiancoders.it/visitor-pattern/> - Visitato settembre 2022
- [10] InfoWorld – *How to work with log4net in C#*
<https://www.infoworld.com/article/3120909/how-to-work-with-log4net-in-c.html> - Visitato settembre 2022
- [11] Code Project – *A brief introduction to the log4net logging library, using C#*
<https://www.codeproject.com/Articles/8245/A-Brief-Introduction-to-the-log4net-logging-librar> - Visitato settembre 2022
- [12] Wikipedia – *Windows Presentation Foundation*
https://it.wikipedia.org/wiki/Windows_Presentation_Foundation - Visitato settembre 2022
- [13] Microsoft – *XAML overview (WPF .NET)*
<https://docs.microsoft.com/en-us/dotnet/desktop/wpf/xaml/?view=netdesktop-6.0> - Visitato settembre 2022
- [14] Code Project – *ICommand Interface in WPF*
<https://www.codeproject.com/Articles/1052346/ICommand-Interface-in-WPF> - Visitato settembre 2022

Indice Figure

Figura 1 – Schema semplificativo progetto	7
Figura 2 – Esempio utilizzo ProcessStartInfo	17
Figura 3 – Esempio struttura parte iniziale template XML	19
Figura 4 – Esempio struttura parte centrale template XML	20
Figura 5 – Esempio struttura parte finale template XML	21
Figura 6 – Parte iniziale metodo FindUsingLibs	22
Figura 7 – Metodo AddNugetSource	22
Figura 8 – Metodo ImportPacket	23
Figura 9 – Esempio d’uso della classe CommandLineParser	24
Figura 10 – Schema funzionamento Deserializzazione	25
Figura 11 – Classe Deserializer	26
Figura 12 – Schema funzionamento Serializzazione	26
Figura 13 – Metodo SerializeIntoCSProj	28
Figura 14 – Esempio d’uso Visitor	29
Figura 15 – Schema UML Visitor	31
Figura 16 – Classi che usano Visit	31
Figura 17 – Schema file contenuti nella cartella compressa	33
Figura 18 – Metodo JSONString	33
Figura 19 – Esempio parte iniziale file map.json	34
Figura 20 – Esempio file manifest.json	34
Figura 21 – Utilizzo dei log nel progetto	37
Figura 22 – Interfaccia iniziale GUI	40
Figura 23 – Esempio d’uso RelayCommand	42
Figura 24 – Esempio aggiornamento campo	42
Figura 25 – Esempio serializzazione in GUI	43

Figura 26 – Sezione “General” GUI	43
Figura 27 – Sezione “Input” GUI	44
Figura 28 – Sezione “Type” GUI	45
Figura 29 – implementazione classe SVGControl	46
Figura 30 – Sezione “View” GUI	47
Figura 31 – Confronto HEX di due file XML	48

Ringraziamenti

Un sentito ringraziamento va al mio relatore, il professore Ciaccia Paolo per la sua disponibilità. Ci tengo soprattutto inoltre a ringraziare Mattia che mi ha permesso di iniziare il percorso di tirocinio e tesi in azienda e che mi ha aiutato con le sue idee e spunti di riflessione, si è sorbita la mia faccia perplessa ogni volta che mi chiedeva di modificare il progetto, ma ha comunque saputo aiutarmi rispiegandomi la sua visione. Il grazie più grande va a mia madre e a mio padre per come mi hanno educato e sostenuto in ogni momento. Grazie per non avermi permesso di arrendermi, anche quando volevo farlo perché vedevo gli esami troppo difficili da sostenere. Al pari dei miei genitori devo ringraziare Bianca, mia sorella, che ha visto più potenziale in me di nessun'altra persona io conosca e mi ha spronato ad iniziare questo percorso universitario. Senza voi tre, tutto questo non sarebbe stato possibile. Questo spazio vorrei poi dedicarlo anche alle persone che sono state con me in questi anni, magari anche solo per un breve periodo. Grazie ad Carmela, Youssef, Alice, Emanuela e Bleona per avermi fatto sentire a casa nel primo anno accademico. Mi spiace che le nostre strade si siano divise ma vi porterò sempre nel cuore. Grazie a Paula, Ahmad, Adriana, Silvia e Maria, la mia "famiglia internazionale" conosciuta in Erasmus. Siamo diventati così tanto in così poco tempo. Il mio Erasmus non sarebbe stato lo stesso senza di voi. Infiniti ringraziamenti andrebbero a tutti i miei compagni di università e a tutte quelle persone che hanno incrociato la loro vita con la mia e che mi hanno lasciato qualcosa. Grazie a Caterina, Zak, Anna, Alice, Steve, Serena e Rov per essere stati complici, ognuno a suo modo, in questo percorso. Grazie per aver ascoltato i miei sfoghi e per tutti i momenti di spensieratezza condivisi insieme.