Università di Genova | DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

SCUOLA DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di laurea in INFORMATICA (L-31)

Anno accademico 2024/2025

Progettazione e realizzazione di un sistema Software per la mappatura degli elementi di un impianto ferroviario

Candidato: Perricone Giuseppe

Relatore
Prof. Rovetta Stefano

Co – Relatore
Fabrizio Galletto

Co – Tesista
Bestoso Gabriele

Sommario

Abstract

Nei moderni sistemi ferroviari, garantire la coerenza e l'affidabilità delle informazioni è fondamentale per assicurare la sicurezza operativa e il corretto funzionamento degli impianti. Tuttavia, strumenti differenti possono produrre rappresentazioni non coincidenti della stessa configurazione di stazione, con il rischio di introdurre discrepanze nelle fasi di progettazione o manutenzione.

Il software sviluppato in questo elaborato prende in input due file XML prodotti da tool diversi e contenenti configurazioni della medesima stazione, analizzandone la struttura e i dati con l'obiettivo di individuare incoerenze ed elementi non corrispondenti.
L'output generato consiste in un report dei risultati, un file di log delle operazioni effettuate e un file di mappatura tra gli identificativi numerici degli elementi equivalenti nei due file.

Poiché il contesto applicativo rientra nell'ambito dei sistemi *safety critical*, è stato adottato il principio della *software diversity*: lo stesso algoritmo è stato implementato in modo indipendente in due linguaggi differenti (C# e Python), al fine di ridurre il rischio di errori sistematici. È stata inoltre definita una standardizzazione dell'output, per consentire un confronto coerente tra le due versioni.
Infine, per entrambe le implementazioni è stata sviluppata una suite di test automatici, al fine di validare il comportamento del software rispetto ai requisiti iniziali e aumentarne l'affidabilità complessiva.

1. Introduzione

Nell'ambito della progettazione di sistemi complessi, informatici e non, la gestione corretta e coerente dei dati rappresenta un elemento fondamentale. Dati inconsistenti, incompleti o formattati in modo non uniforme possono compromettere l'affidabilità delle applicazioni e richiedere attività di rielaborazione che incidono sia sui tempi sia sulla qualità del risultato finale. Garantire integrità, coerenza e tracciabilità delle informazioni è quindi essenziale per assicurare il corretto funzionamento dei sistemi e per evitare errori che potrebbero propagarsi nelle fasi successive del processo.

Nel presente elaborato si andrà a trattare il caso specifico dei sistemi ferroviari.
In questo ambito la gestione della configurazione di un impianto rappresenta uno degli aspetti centrali. Infatti, ogni impianto viene descritto da una grande quantità di informazioni strutturate, che devono rimanere coerenti lungo tutto il ciclo di vita del sistema: dalla progettazione alla messa in servizio, fino alle attività di manutenzione ed evoluzione dell'infrastruttura.
Si tratta di un dominio molto complesso, caratterizzato da una quantità di informazioni elevata. La complessità del dominio, unita ai rigorosi requisiti di sicurezza propri del settore, rende essenziale disporre di strumenti affidabili per la verifica e il controllo della consistenza dei dati.

In questo contesto nasce il progetto da me trattato in questo elaborato.
Sono presenti due tool esterni differenti, denominati come TOOL_A e TOOL_B, entrambi producono una documentazione molto dettagliata dell'impianto ferroviario di Stroncone in formato xml. Tuttavia, i due tool operano su due basi di dati differenti generando quindi due file che possiedono la stessa medesima struttura, e quindi compatibili, ma vi è il rischio, che a stessi elementi dell'impianto vengano assegnati identificativi numerici, e dettagli generali, differenti tra i due file. Ciò è molto pericoloso,

perché può portare a discrepanze e incoerenze nella documentazione, con potenziali ripercussioni sulle attività di manutenzione e, più in generale, sull'efficienza e la sicurezza dell'infrastruttura.

L'obiettivo del lavoro presentato è quindi la progettazione e lo sviluppo di un sistema software, implementato in linguaggio C#, in grado di confrontare le due configurazioni e di individuare eventuali incongruenze. Il software, in particolare, deve essere in grado di:

- Analizzare in input i due file XML contenenti le configurazioni dell'impianto;
- Andare comparare in modo efficiente e sicuro i due file;
- Generare un report testuale che contenga tutte le varie problematiche che sono state individuate
- Generare un file di log che contenga tutte le operazioni che vengono svolte all'interno del programma. Ciò serve per garantire quella che viene definita come tracciabilità delle informazioni
- Generare un file di mapping, che metta in corrispondenza gli identificativi numerici dei vari elementi, facilitando il confronto e la consultazione umana dei dati.

Dopo la realizzazione del software è stato affrontato anche l'aspetto legato al contesto *safety critical* in cui esso si inserisce.

A differenza di molti ambiti di progettazione software, l'ambito ferroviario opera in un contesto dove sono richiesti livelli molto elevati di affidabilità, in quanto eventuali incoerenze o malfunzionamenti potrebbero avere conseguenze rilevanti sul piano operativo e sulla sicurezza di terzi. In questo contesto è stata messo sotto osservazione il concetto di *software diversity*, secondo cui lo stesso algoritmo viene implementato su differenti ambienti di sviluppo, quindi diversi linguaggi di programmazione, piattaforme o stili implementativi, con lo scopo di produrre in output gli stessi medesimi risultati in modo tale da ridurre la probabilità di errori comuni o

correlati. A tal fine, partendo dal progetto originariamente sviluppato in C# dal collega Bestoso Gabriele, è stata realizzata in maniera completamente autonoma una seconda implementazione in linguaggio Python, basata unicamente sulle medesime specifiche funzionali richieste.

Completata la re-implementazione, è stata effettuata una fase di verifica del “doppio canale”, confrontando gli output prodotti dalle due versioni del software. Per facilitare tali confronti e consentire eventuali controlli automatici futuri, è stato definito e applicato un formato standardizzato di output, così da garantire uniformità nei risultati e facilitarne la confrontabilità.

Infine, per entrambe le implementazioni è stata sviluppata una suite di unit test, rispettivamente in C# e Python, con l’obiettivo di verificare il rispetto dei requisiti funzionali e aumentare l’affidabilità complessiva del sistema.

2. Contesto Teorico

Questo capitolo ha l’obiettivo di presentare il contesto in cui si inserisce il lavoro svolto, fornendo una panoramica sugli elementi principali che compongono un impianto ferroviario, tra questi ci si soffermerà maggiormente sugli elementi con cui abbiamo lavorato.
Verrà inoltre presentato il formato XML utilizzato per la rappresentazione delle configurazioni, mostrando come le informazioni relative all’impianto vengono strutturate e organizzate, all’interno di questi file. Infine, saranno introdotti i concetti di *diversity* e *safety critical* all’interno del contesto ferroviario.
(Le foto, e la maggior parte dei concetti teorici, sono stati reperiti dal “*Gilardi”* [1])

2.1 Struttura di un Impianto ed Elementi Caratteristici

La rappresentazione di un impianto ferroviario si basa su quello che viene definito *piano schematico.*

Per piano schematico si intende una rappresentazione schematica dell'impianto, che mette in evidenza la disposizione dei vari elementi e la loro relazione funzionale. Non si parla quindi della loro rappresentazione fisica, ma della logica effettiva dell'impianto.

In altre parole, quindi il piano schematico descrive:

- Quali elementi compongono l'impianto (ad esempio binari, scambi, segnali, circuiti di binario)
- Come questi elementi sono collegati tra loro
- Quali caratteristiche e vincoli li contraddistinguono

Questo tipo di rappresentazione costituisce la base a partire dalla quale vengono generati i file XML prodotti dai tool analizzati in questa tesi. All'interno di tali file, ogni elemento dell'impianto viene codificato attraverso identificativi numerici, attributi e relazioni, in modo da consentire al software di interpretarli in maniera univoca e strutturata.

Un impianto ferroviario è composto da vari elementi funzionali, i quali cooperano tra di loro per garantirne il corretto funzionamento, in modo sicuro.

Di seguito verranno descritte le principali categorie di interesse, tra queste alcune hanno un approfondimento maggiore di altre perché sono quelle direttamente coinvolte nello sviluppo del progetto descritto nell'elaborato, mentre altre sono aree che non sono state toccate:

2.1.1 Deviatoi

Il deviatoio è l'elemento dell'infrastruttura ferroviaria che consente il passaggio di un treno da un binario a un altro. È, in sostanza, ciò che permette la biforcazione del

percorso, rendendo possibile l'instradamento dei convogli all'interno di una stazione o lungo una linea.

Dal punto di vista funzionale, un deviatoio può essere o in posizione *normale*, dove quindi il treno prosegue sul suo percorso principale, oppure in posizione *rovescia*, dove il treno viene instradato verso il ramo deviato

La Figura 1 mostra in modo semplificato la struttura logica di un deviatoio: il ramo *a* rappresenta la prosecuzione diretta del binario, mentre il ramo *b* rappresenta la derivazione verso un altro binario.

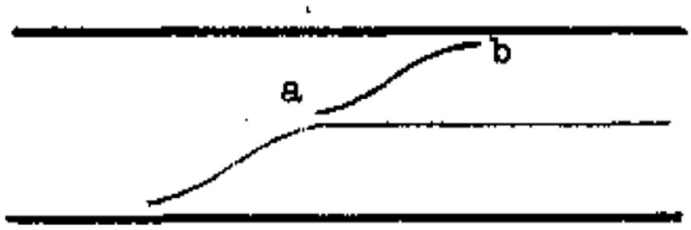

Figura 1: Rappresentazione schematica di un deviatoio

Per ragioni di sicurezza, la posizione del deviatoio deve essere monitorata e coerente con gli itinerari impostati: un instradamento non compatibile con la posizione del deviatoio potrebbe infatti condurre a situazioni pericolose.

Nel contesto di questo elaborato, la corretta identificazione e corrispondenza dei deviatoi all'interno del file XML è importantissima, dal momento che il software sviluppato deve confrontare i vari elementi, ma provenienti da file XML differenti, identificandone eventuali incongruenze. Le modalità con cui vengono identificati i deviatoi all'interno del file XML sono approfondite all'interno della sezione 2.2.

2.1.2 Circuiti di Binario (CDB)

I circuiti di binario sono elementi essenziali per la sicurezza all'interno del contesto ferroviario, in quanto permettono di rilevare la presenza di treni, o altri oggetti, su un

tratto di binario. Ogni linea viene quindi divisa in più sezioni isolate, ciascuna viene successivamente associata ad un circuito, che segnala se la sezione è:

- Libera: nessun veicolo è presente sul tratto;
- Occupata: in questo caso viene rilevata la presenza del veicolo, o di qualsiasi altra anomalia fisica. Il tutto notificato al sistema di controllo

La figura 2 mostra un esempio semplificato di suddivisione di una linea in più circuiti di binario, ciascuno dei quali deve coprire una porzione sufficiente di binario per garantire un rilevamento sicuro e continuo della presenza dei veicoli.

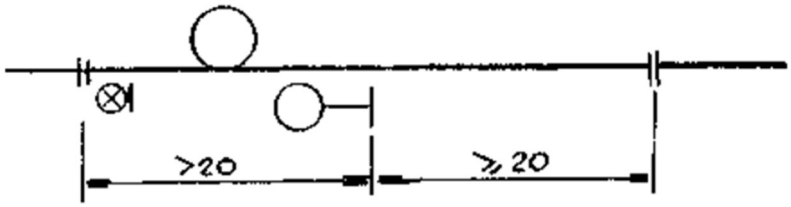

Figura 2: Rappresentazione schematica di una linea suddivisa in circuiti di binario

La presenza o meno di un treno su un CdB è un'informazione cruciale per la gestione della circolazione:
un CdB occupato impedisce l'impostazione di itinerari che includono quel tratto, prevenendo così movimenti non autorizzati o collisioni.
Nel contesto di questa tesi, i circuiti di binario costituiscono uno degli elementi principali sottoposti a confronto tra i due file XML generati dai tool analizzati.
La modalità con cui i circuiti di binario vengono rappresentati e descritti nei file XML sarà illustrata in dettaglio nella Sezione 2.2.

2.1.3 Itinerari

Un itinerario rappresenta un percorso prestabilito che un treno può compiere all'interno di un impianto ferroviario, tra un punto di origine e un punto di arrivo.

Non si tratta solamente di una sequenza di binari, ma di una vera e propria configurazione logica che tiene in considerazione la presenza di più scenari in modo da garantire uno spostamento del treno in sicurezza.
La figura 3 mostra in modo molto schematico la struttura dei binari di una stazione. Questa rappresentazione non raffigura il singolo itinerario, ma il contesto generale in cui andiamo a definirlo; ogni itinerario che possiamo generare a partire da questa immagine è infatti un “sottoinsieme” di questo schema, in funzione della direzione di movimento, della direzione degli scambi, e molto altro.

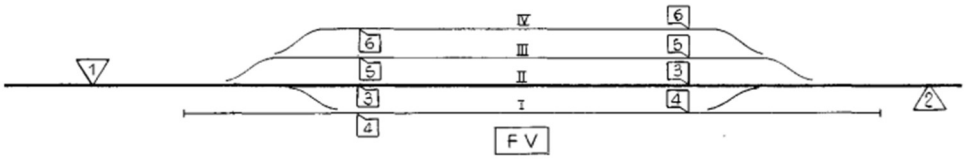

Figura 3: Schema semplificato di una stazione

All’interno di questo elaborato non verranno espressi tutti gli elementi e le notazioni tecniche relative agli itinerari, poiché non sono oggetto di ciò che è stato sviluppato; tuttavia, possiamo affermare che i punti chiave che definiscono un itinerario sono:
La presenza di un segnale di origine (o di partenza), una sequenza di circuiti di binario che compongono il percorso, e la posizione dei deviatoi lungo tale percorso.
Solo quando tutti i circuiti di binario sono liberi, e tutti i deviatoi risultano posizionati correttamente, l’itinerario può essere impostato e il movimento autorizzato.
Nel contesto di questa tesi, gli itinerari sono utilizzati nel progetto implementato in Python, dove sono rappresentati in forma strutturata all’interno di un file XML dedicato. La descrizione dettagliata del relativo formato XML è riportata nella Sezione 2.2.

In un impianto ferroviario reale sono presenti ulteriori elementi, quali segnali luminosi, instradamenti, sistemi di protezione e altri dispositivi di sicurezza, che contribuiscono alla gestione completa della circolazione. Tuttavia, tali aspetti non rientrano

direttamente nell'ambito applicativo del progetto descritto in questa tesi e pertanto non verranno approfonditi.
Nei paragrafi successivi verrà invece illustrato come gli elementi sopra descritti (deviatoi, circuiti di binario e itinerari) vengono rappresentati all'interno dei file XML, evidenziando le differenze strutturali tra i due tool presi in esame.

2.2 Gestione della Configurazione XML di un Impianto

Partendo dal piano schematico si può generare la configurazione logica dell'impianto ferroviario, ovvero l'insieme delle informazioni che descrivono come i suoi elementi sono organizzati e collegati tra di loro.
Nell'ambito di questo elaborato, tale configurazione viene rappresentata all'interno di un file in formato XML (eXtensible Markup Language).
L'utilizzo di questo formato permette di descrivere l'impianto in modo strutturato e leggibile, preservando tutte le informazioni importanti. Inoltre, è un formato dati particolarmente adatto alla gestione automatizzata tramite software, poiché permette di accedere ai dati in modo molto semplice ed efficiente.
All'interno dei file XML, ciascun componente dell'impianto è descritto come una singola entità, inserita in una struttura gerarchica ad albero. Ogni entità è caratterizzata da attributi e riferimenti ad altre entità, consentendo di rappresentare un impianto in maniera chiara e univoca. Questa univocità risulta fondamentale perché permette di semplificare notevolmente il confronto con altre configurazioni, a condizione che lo schema e le modalità di rappresentazione adottate siano compatibili.

Nel caso dei CdB e dei deviatoi, la rappresentazione XML distingue tra due concetti principali, ovvero l'entità "*Ente"* e l'entità "*Nodo".*

Un'entità "*ente*" rappresenta un elemento funzionale dell'impianto, mentre un "*nodo*" rappresenta un punto di connessione logica tra più entità, descrivendo come sono collegate.

Ogni ente e ogni nodo posseggono più attributi, tra cui la categoria, che aiuta ad identificare il tipo di oggetto a cui l'entità si riferisce.

Nel contesto di questo elaborato, l'analisi si concentra in particolare su nodi di categoria deviatoio ed enti di tipo CdB. Le figure 4 e 5 contengono esempi di come sono rappresentati enti e nodi nel formato XML.

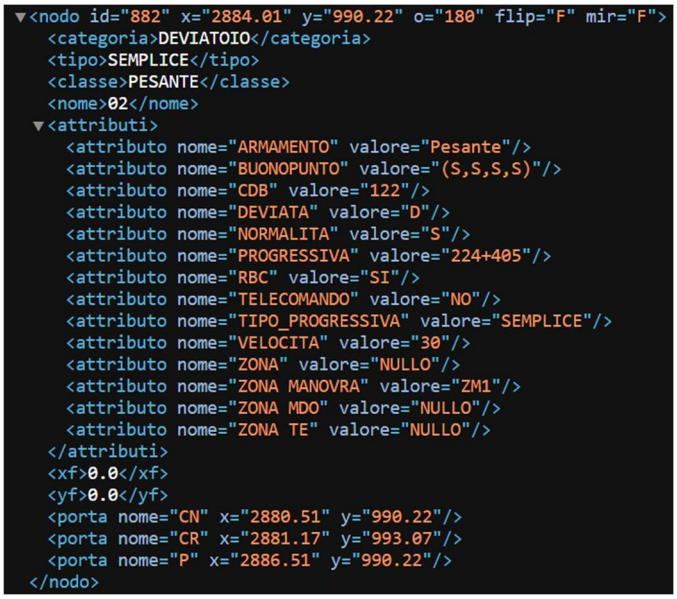

Figura 4: Esempio di nodo XML

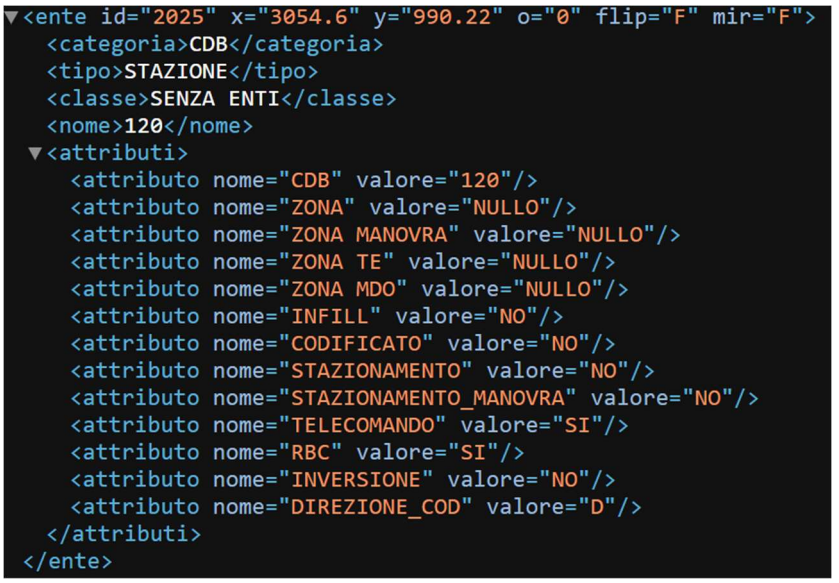

Figura 5: Esempio di ente XML

Nella sezione precedente, oltre che di deviatoi e CdB, sono stati introdotti anche gli itinerari. A differenza dei precedenti, gli itinerari, non sono descritti all'interno dello stesso file XML, ma in un altro apposito. Questo perché gli itinerari rappresentano quelle che sono delle configurazioni logiche in movimento, ovvero percorsi effettivi che il treno può percorrere. Nel contesto di questo elaborato gli itinerari sono stati analizzati e sfruttati nel secondo progetto, ovvero quello che ho implementato in Python; per maggiori dettagli funzionali si rimanda quindi alla sezione dedicata.

A livello XML un itinerario possiede una struttura molto diversa da quelle appena visionate:

ogni itinerario, infatti, è rappresentato come entità IXLItem a cui sono associate:

- Una lista di "*TrackCircuit*" (Circuiti di binario)
- Una lista di "*PointList"* (per i deviatoi coinvolti)
- Nome dell'impianto e liste di nodi, e altri elementi. Tuttavia quest'ultimi non sono stati usati in ambito progettuale

Figura 6: Esempio di itinerario XML

2.3 Concetto di Diversity nei sistemi Safety Critical

All'interno del contesto ferroviario, molti sistemi software operano nella categoria dei sistemi noti come *safety critical,* ovvero tutti quei sistemi il cui malfunzionamento può avere delle conseguenze rilevanti sulla sicurezza di soggetti terzi, sull'integrità delle infrastrutture stesse o sulla continuità operativa del servizio.

In questo ambito non è sufficiente solo che il software "funzioni", ma è necessario garantire un livello molto elevato di affidabilità, riducendo al minimo la possibilità di guasti, errori logici o comportamenti anomali.

Per questo motivo lo sviluppo software in questi ambiti è regolato da regole e standard specifici, tra questi le norme che interessano ai fini dell'elaborato sono CENELEC EN

50128 (per lo sviluppo software ferroviario) ed EN 50129 (per la sicurezza dei sistemi di segnalamento e controllo), le quali definiscono processi, criteri di validazione e metodologie per assicurare la sicurezza operativa.
Tra le tecniche raccomandate da tali standard vi è il principio di *software diversity*.

Per *software diversity* si intende l'adozione di più implementazioni indipendenti l'una dall'altra dello stesso algoritmo o della stessa funzione, realizzate con:

- Linguaggi di programmazioni differenti
- Ambienti di sviluppo differenti
- Processi di progettazione separati

L'obiettivo è cercare di migliorare l'affidabilità generale del sistema, andando a ridurre la probabilità di errori derivanti dalla fase di progettazione, e non dal normale degrado del SW; quindi, da utilizzo di SW datati oppure librerie non aggiornate.
Se un errore appare in una sola delle due implementazioni, l'altra funge da riferimento per l'individuazione dell'anomalia, aumentando così la robustezza complessiva del sistema.
All'interno di questo elaborato il principio di diversity è stato applicato adottando il *doppio canale* ("*dual channel"*), ovvero l'implementazione di due versioni indipendenti dello stesso software:
una in linguaggio C# (implementata dal co-tesista Bestoso Gabriele) e una in Python (implementata da me). Le due versioni vengono eseguite parallelamente con lo scopo di confrontare in tempo reale i risultati prodotti.

2.4 Convalida del Doppio Canale

L'obiettivo del doppio canale non è quello di "raddoppiare" la potenza computazionale, ma di mitigare gli errori sistematici. Se le due implementazioni indipendenti producono **lo stesso output**, è possibile considerare il risultato coerente rispetto alle specifiche.

Qualora invece si rileva una discrepanza, questa costituisce un segnale d'allarme che richiede analisi approfondita.

La convalida incrociata è quindi fondamentale per ridurre al minimo gli errori software che possono essere prodotti, e l'implementazione del doppio canale è una delle strategie migliori per raggiungere questo fine ultimo.
Per rendere efficace la convalida incrociata è fondamentale che i risultati siano comparabili.
A tal fine è stato definito e adottato un formato di output standardizzato, che stabilisce:

- quali file generare,
- la loro nomenclatura,
- la struttura interna dei contenuti.

In questo modo, il confronto tra gli output delle due versioni può essere eseguito in modo semplice, affidabile e potenzialmente automatizzabile, riducendo il rischio di ambiguità interpretative.
L'applicazione congiunta di queste tecniche ha permesso di ottenere due implementazioni che producono un output pressoché identico, validato e verificato rispetto ai requisiti iniziali, garantendo così un livello elevato di affidabilità del sistema.

3. Analisi e Metodologie di Sviluppo (Progetto C#)

In questo capitolo vengono descritte le principali metodologie adottate nello sviluppo del progetto software realizzato in linguaggio C#. L'obiettivo non è quello di presentare nel dettaglio l'implementazione o le singole scelte architetturali, ma di illustrare l'approccio metodologico adottato, motivandone la selezione in relazione agli obiettivi di affidabilità, manutenibilità e coerenza del sistema.
Verranno quindi introdotti i criteri utilizzati per l'analisi dei requisiti, le tecniche adottate per la gestione e la mappatura dei dati contenuti nei file XML, il ruolo del file di

configurazione, le considerazioni relative alla sicurezza del software, le attività di refactoring e infine la progettazione della suite di unit test.
Per una descrizione approfondita dell'architettura del software e del flusso operativo dell'applicazione si rimanda al Capitolo 4.

3.1 Analisi dei Requisiti

Come nella quasi totalità dello sviluppo dei progetti software, la prima fase è quella dell'analisi dei requisiti del sistema che deve essere implementato. Questa fase è molto importante, perché permette di definire in modo chiaro e verificabile che cosa deve fare il software, e fare in modo che non vi sia ambiguità nelle sue specifiche.
Possiamo distinguere i requisiti in due categorie principali:

i *requisiti funzionali,* ovvero quelli che descrivono le operazioni e le funzionalità che il software deve essere in grado di svolgere. Nel caso specifico del progetto descritto in questo elaborato un esempio di requisiti funzionali possono essere, la capacità del sistema di leggere e interpretare i file XML in ingresso, confrontarne in modo strutturato il contenuto, identificare eventuali discrepanze e produrre gli output richiesti.

Abbiamo poi i *requisiti non funzionali,* ovvero quelli che descrivono come il software deve comportarsi, includendo quindi dettagli che non incidono sulle operazioni, ma bensì sulle prestazioni, sulla sicurezza e sull'architettura del sistema.
Sulla base dell'analisi iniziale del problema e delle specifiche fornite, i requisiti individuati per il progetto sono riportati nelle seguenti tabelle (Figura 7 e Figura 8):

	Requisiti di Sistema
SRS_XML_010	Il sistema deve leggere la rappresentazione della struttura di un impianto ferroviario generata da TOOL_A e TOOL_B in un file XML
SRS_CONF_010	Il sistema deve accettare in input un file di configurazione esterno per specificare l'insieme di elementi da mappare
SRS_MAP_010	Il sistema deve generare un file di mapping che colleghi tra loro gli identificativi numerici degli stessi elementi presenti nei due file XML
SRS_MAP_020	Il sistema deve permettere la creazione del mapping solo per gli elementi indicati nel file di configurazione
SRS_CONF_020	Il sistema deve verificare la corrispondenza effettiva tra gli elementi mappati nei due file XML
SRS_REP_010	Il sistema deve generare un report dettagliato con gli esiti della verifica effettuata sugli elementi mappati (abbiamo ipotizzato che il report venga generato in automatico) in formato testuale
SRS_LOG_010	Il sistema deve fornire i log con le varie azioni che vengono effettuate

Figura 7: Tabella dei requisiti funzionali

SAS_ARCH_010	Il sistema deve essere composto da quattro moduli: un modulo di lettura da XML, un modulo di gestione del file di configurazione, un modulo di gestione del file di mapping e un modulo di scrittura report e file di log
SAS_MOD_XML_010	Il modulo di lettura da XML deve usare una libreria LINQ to XML per leggere la struttura dei file
SAS_MOD_CONF_010	Il modulo di gestione del file di configurazione deve caricare il file per filtrare gli elementi da mappare
SAS_MOD_MAP_010	Il modulo di gestione del file di mapping deve confrontare gli identificativi numerici degli elementi indicati e generare un file di mapping
SAS_MOD_REP_010	Il modulo di report deve generare un file contenente il risultato del confronto e delle verifiche di corrispondenza
SAS_MOD_LOG_010	Il modulo di log deve generare un file log per tener traccia delle operazioni effettuate

Figura 8: Tabella dei requisiti non funzionali

Successivamente al loro sviluppo le tabelle dei requisiti sono state sottoposte a una fase di revisione con i referenti aziendali, al fine di verificarne la completezza e

l’aderenza alle necessità operative reali. Solo al termine di tale validazione si è proceduto alla fase successiva di progettazione e implementazione del software.

3.2 Analisi del problema

Prima della fase di sviluppo è stata condotta un’analisi approfondita della struttura dei file XML in ingresso, al fine di comprendere come le informazioni relative all’impianto ferroviario fossero organizzate e rappresentate all’interno dei due sistemi. L’obiettivo principale era individuare un metodo di confronto affidabile, capace di riconoscere correttamente gli stessi oggetti in entrambe le configurazioni, anche in presenza di differenze a livello di identificativi.

I due file XML prodotti da TOOL_A e TOOL_B seguono lo stesso modello logico e contengono lo stesso insieme di entità (deviatoi, circuiti di binario, segnali, nodi, ecc.). Tuttavia, poiché vengono generati a partire da basi di dati diverse, gli identificativi numerici associati ai medesimi elementi dell’infrastruttura possono differire tra un file e l’altro. Inoltre, durante l’analisi è emerso che le differenze non si limitano ai soli ID, ma possono riguardare anche proprietà interne o attributi descrittivi delle entità.
Di conseguenza, un confronto diretto basato esclusivamente sugli identificativi risulterebbe inaffidabile, poiché elementi logicamente equivalenti potrebbero essere classificati come differenti o viceversa.

Per risolvere questo problema è stato necessario definire una strategia di mappatura che non si basasse su un singolo attributo, ma su un insieme coerente di proprietà e caratteristiche in grado di distinguere ogni elemento in modo univoco.
A tal fine è stato introdotto un *file di configurazione*, costruito durante l’analisi e non fornito inizialmente. Il suo scopo è quello di guidare il processo di mappatura indicando, per ciascuna tipologia di elemento, quali proprietà devono essere

considerate nel confronto. Il *configuration file* rappresenta quindi il punto di riferimento che stabilisce come identificare gli elementi e come valutarne la corrispondenza tra i due sistemi.

3.3 Il File di Configurazione

Il file di configurazione, introdotto nella sezione precedente, svolge il ruolo di riferimento principale per guidare il processo di mappatura tra gli elementi presenti nei due file XML. Al momento della consegna del progetto, era noto che tale file fosse previsto, ma non erano state fornite indicazioni su come dovesse essere strutturato né su quali informazioni dovesse contenere. È stato quindi necessario definirne in autonomia sia il formato che la logica interna.

Durante l'analisi preliminare dei due file XML, è stato osservato che, pur differendo gli identificativi numerici degli elementi, e altri attributi, ve ne sono tuttavia alcuni che risultano in entrambe le configurazioni. Questi attributi costanti possono quindi essere utilizzati come riferimento stabile per riconoscere lo stesso oggetto all'interno dei due sistemi, fornendo la base necessaria per definire una mappatura affidabile.
Sulla base di questa osservazione è stato deciso di costruire il file di configurazione in formato XML, allo scopo di mantenere omogeneità con i file sorgente e facilitare l'elaborazione. Per ciascuna tipologia di elemento (ad esempio deviatoi, segnali o circuiti di binario), il file elenca gli attributi da utilizzare nella fase di identificazione, ovvero quelli che restano coerenti in entrambi i file, quali:

- *Categoria* (es. Deviatoio)
- *Nome* (es. 02)
- *Classe* (es. pesante)
- *Tipo* (es. semplice, senza enti, ecc)

Il file di configurazione costituisce quindi un livello intermedio tra la struttura astratta del dominio ferroviario e l'implementazione software del confronto, rendendo il sistema

più flessibile, estensibile e controllabile.

Una modifica ai criteri di identificazione, infatti, non richiede di intervenire direttamente sul codice, ma semplicemente di aggiornare il file di configurazione.

Dal punto di vista operativo, il file viene letto nella fase iniziale dell'esecuzione del programma e ne guida il comportamento durante l'intero processo di analisi.

La figura 9 contiene un esempio del file di configurazione:

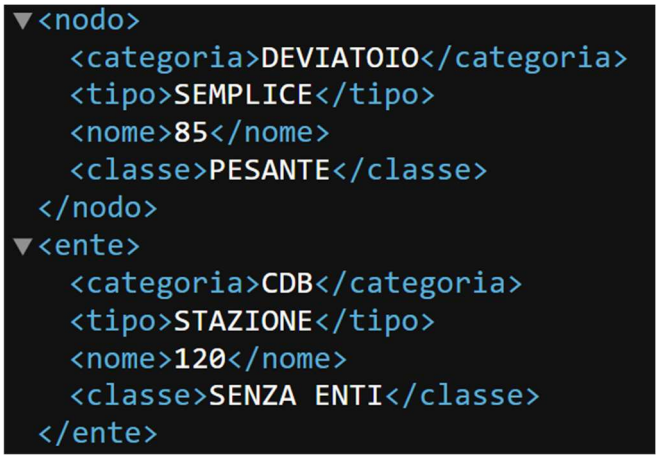

Figura 9: Rappresentazione di nodo ed ente nel file di configurazione

3.4 Alcune Considerazioni di Sicurezza – File Browsing

Una volta definite le modalità di mappatura e costruito il file di configurazione, è stato possibile avviare lo sviluppo operativo del software. Durante questa fase è emersa la necessità di tenere in considerazione alcuni aspetti legati alla sicurezza e all'affidabilità del sistema, in particolare per quanto riguarda la gestione dei file di input e di output.

In una prima versione del progetto, i percorsi dei file XML da analizzare e dei file generati in uscita erano inseriti direttamente nel codice sotto forma di stringhe fisse. Questa soluzione, seppur funzionale in fase di prototipazione, presenta diversi svantaggi: riduce la flessibilità del software, aumenta il rischio di errori in fase di aggiornamento e rappresenta una cattiva pratica di progettazione, poiché vincola l'esecuzione a percorsi specifici del file system. Per questo motivo, a seguito di una

revisione con il tutor aziendale, si è deciso di sostituire tale approccio con una procedura di *file browsing*, che permette all'utente di selezionare manualmente i file XML da usare come input, il file di configurazione e di scegliere anche la cartella all'interno della quale verranno salvati i file di output (report, log e mapping).
Nella fase di selezione il programma effettua anche delle verifiche preliminari, verificando che il file scelto sia compatibile, e che il tutto venga selezionato in maniera corretta. In questo modo si garantisce che vengano elaborati solo file corretti e coerenti con il dominio applicativo.

Parallelamente, è stata introdotta una classe dedicata alla gestione del log, che registra ogni operazione significativa eseguita dal programma. La presenza di un file di log consente di mantenere una piena tracciabilità del processo di confronto, permettendo di ricostruire eventuali problemi e di verificare l'aderenza del comportamento del software ai requisiti.
L'insieme di questi accorgimenti non mira tanto alla sicurezza in senso informatico tradizionale (ad esempio protezione da accessi esterni), quanto piuttosto a garantire un comportamento affidabile e controllabile del software, requisito fondamentale nel contesto operativo in cui esso si inserisce.

3.5 Refactoring del Codice

Durante lo sviluppo del progetto, una volta implementata una prima versione funzionante del software, è stata effettuata una fase di *refactoring*, ovvero un insieme di attività volte a migliorare la struttura interna del codice senza modificarne il comportamento esterno.
Lo scopo del refactoring non è quindi aggiungere nuove funzionalità, ma rendere il software più chiaro, più semplice da mantenere, più modulare ed estensibile, riducendo al contempo il rischio di introdurre errori in fasi successive.

Il refactoring rappresenta quindi una pratica essenziale nello sviluppo software professionale, poiché consente di:

- Migliorare la leggibilità;
- Ridurre la complessità;
- Ottenere un software più facile da mantenere;
- Raggiungere delle prestazioni più elevate;
- Le estensioni diventano più semplici da sviluppare;

Nel caso specifico del progetto sviluppato in C#, il refactoring ha riguardato principalmente i seguenti aspetti:

1) *Separazione delle Responsabilità*

Nelle prime versioni del software alcune funzionalità diverse erano concentrate all'interno delle stesse classi o metodi.
Le funzionalità sono state suddivise in moduli più chiari e indipendenti (es. parsing XML, confronto logico, gestione log, scrittura dell'output), in linea con il principio *Single Responsibility Principle (SRP).*

2) *Eliminazione di Codice Duplicato*

Durante lo sviluppo iniziale alcune operazioni venivano ripetute in più punti del codice, con lo stesso blocco di istruzioni replicato.
Queste parti sono state isolate in metodi specifici e riutilizzabili, riducendo la ridondanza e semplificando la manutenzione.

3) *Miglioramento della Leggibilità e Consistenza*

Sono state eseguite attività di rinomina di variabili, classi e metodi, suddivisione di metodi troppo lunghi e riorganizzazione dei file, così da rendere il flusso logico più chiaro e immediatamente interpretabile.

4) Semplificazione della Logica di Confronto

Nella prima versione, la logica di confronto tra gli elementi dei due XML era implementata con strutture condizionali complesse e annidate.

Questa parte è stata ristrutturata per rendere più lineare il processo di mappatura, sfruttando funzioni dedicate e l'ausilio del file di configurazione come guida centrale per il confronto.

Grazie a questa fase di refactoring, la qualità generale del software è migliorata notevolmente.

3.6 Unit Testing: Principi Fondamentali

Nel contesto dello sviluppo software, l'attività di testing rappresenta uno strumento essenziale per garantire la qualità, l'affidabilità e la correttezza del codice prodotto. In particolare, gli *Unit Test* consentono di verificare il comportamento delle singole unità logiche del programma (ad esempio, metodi o classi) in modo isolato, controllando che, a partire da determinati input, venga restituito l'output atteso.

L'obiettivo principale dell'unit testing è identificare tempestivamente eventuali malfunzionamenti, riducendo il costo e la complessità della loro risoluzione nelle fasi successive del ciclo di sviluppo.

Nel progetto descritto in questo elaborato, gli unit test sono stati introdotti in una fase avanzata dello sviluppo. Questa scelta non rappresenta la metodologia ideale, poiché ha reso necessario apportare alcune modifiche successive al codice già implementato. Una pratica più corretta avrebbe previsto la creazione graduale dei test in parallelo all'implementazione dei moduli, adottando un approccio iterativo di sviluppo e verifica. Tuttavia, poiché il progetto non presentava dimensioni particolarmente elevate, l'impatto di questa scelta è risultato limitato.

3.6.1 Principi chiave dello Unit Testing

Per essere efficace, l'attività di unit testing deve rispettare alcuni principi fondamentali:

- *Isolamento*

  Ogni test deve verificare una sola unità di codice alla volta, evitando dipendenze da parti esterne del sistema (come database, file di sistema o servizi remoti). Questo consente di individuare con precisione l'origine di un eventuale malfunzionamento.

- *Ripetibilità*

  Un test deve produrre lo stesso risultato indipendentemente dal contesto in cui viene eseguito. La ripetibilità garantisce che eventuali anomalie siano attribuibili esclusivamente al codice sotto test e non a fattori esterni.

- *Chiarezza e Specificità*

  Ogni test deve verificare un comportamento ben definito. Test troppo generici risultano difficili da interpretare e non forniscono indicazioni utili in caso di errore.

- *Automazione*

  Gli unit test devono essere eseguibili in modo completamente automatico. Ciò permette di integrarli facilmente nei processi di sviluppo e di esecuzione continua (*Continuous Integration*).

3.6.2 Gestione delle dipendenze e Integrazione nel flusso di sviluppo

Nel caso in cui l'unità sotto test dipenda da componenti esterni (come servizi, file o database), il principio di isolamento verrebbe compromesso. Per ovviare a questo problema si ricorre all'utilizzo di *Mock*: oggetti che simulano il comportamento delle dipendenze reali. L'uso di *mock* permette di: controllare l'ambiente di esecuzione, verificare non solo il risultato dell'elaborazione, ma anche le interazioni (ad esempio

quale metodo è stato chiamato, con che argomenti e quante volte) e anche di evitare effetti collaterali indesiderati.

Nel progetto, i mock sono stati utilizzati per garantire l'indipendenza dei test e rendere riproducibili le condizioni di esecuzione.

Per una descrizione dettagliata del framework utilizzato e della struttura delle test suite si rimanda al Capitolo 4.

4. Implementazione Del Software (C#)

In questo capitolo viene presentata l'implementazione del software sviluppato da me in linguaggio C#, descrivendo le scelte progettuali fatte e l'architettura del sistema sviluppato. Nei capitoli precedenti è stato spiegato cosa deve fare il software e le scelte metodologiche attuate, oltre alle varie analisi; adesso in questo capitolo si entrerà nel dettaglio delle soluzioni adottate a livello di struttura del codice, organizzazione dei moduli e gestione dei dati durante l'esecuzione. Nella prima parte verrà illustrata l'architettura generale del software, evidenziando come le diverse responsabilità siano distribuite tra i vari componenti. Successivamente verrà descritto il flusso operativo principale, dalla lettura dei file XML in ingresso fino alla generazione dei file di report, log e mapping.

Infine, verranno presentati gli aspetti più rilevanti relativi ai test e alla validazione, con riferimento alle suite di unit test sviluppate per verificare il corretto comportamento del sistema.

4.1 Architettura del Sistema Sviluppato

Come descritto nei capitoli precedenti, il software ha il compito di confrontare due file XML contenenti la descrizione di un impianto ferroviario, individuare gli elementi comuni e quelli divergenti, e generare in output diversi file di risultato (report, log e file di mapping). Per raggiungere questo obiettivo, il sistema è stato suddiviso in moduli con responsabilità ben definite. La prima versione del programma era sviluppata in

modo più compatto e monolitico. A seguito di attività di refactoring successive e review varie, è stato possibile separare in modo chiaro le responsabilità, migliorando leggibilità, manutenibilità ed estensibilità del codice. Il flusso di esecuzione è coordinato dalla classe principale, che gestisce la selezione dei file di input, il parsing dei dati, il confronto tra i due XML e la generazione dei risultati finali. Questa organizzazione ha permesso di ottenere componenti **indipendenti ma cooperanti**, riducendo le dipendenze interne e semplificando il processo di testing e aggiornamento del software. Inoltre, poiché l'applicativo verrà utilizzato in ambito aziendale, particolare attenzione è stata posta alla chiarezza e alla comprensibilità del codice, al fine di facilitarne l'adozione futura.

In Figura 10 è riportato il *class diagram* UML dell'architettura complessiva del sistema, dal quale è possibile osservare le relazioni tra i principali moduli:

- *MainClass*

  Coordina l'intero flusso dell'applicazione. Richiede all'utente la selezione dei file XML e della directory di output, inizializza i moduli necessari e invoca i metodi che compongono il processo di confronto.
- *Parser* (implementa IParser)

  Gestisce il parsing dei file XML, estraendo nodi ed enti secondo le categorie definite nel file di configurazione. Contiene inoltre il metodo generico per eseguire il confronto tra le liste provenienti dai due file.
- *XML_Elem*

  Racchiude le classi che modellano le entità fondamentali dell'impianto (ad esempio *Nodo* ed *Ente*), fornendo una rappresentazione strutturata dei dati estratti.
- *Comparators*

  Modulo di supporto che contiene funzioni per la definizione delle chiavi e dei criteri di confronto, utilizzate dal metodo generico presente in *Parser*.

- *FileBrowser* (implementa IFileBrowser)

  Fornisce l'interfaccia per la selezione dei file in input e della directory di output. Questa soluzione elimina la presenza di percorsi hard-coded nel codice e migliora la sicurezza e la portabilità del software.

- *Mapping*

  Si occupa della generazione del file di mappatura, che associa gli elementi equivalenti tra i due XML.

- *Logger*

  Registra tutti gli eventi significativi che avvengono durante l'esecuzione del programma, consentendo tracciabilità e analisi di eventuali anomalie.

- *FileWrite*

  Gestisce la scrittura dei file di risultato (report), garantendo un formato consistente e leggibile.

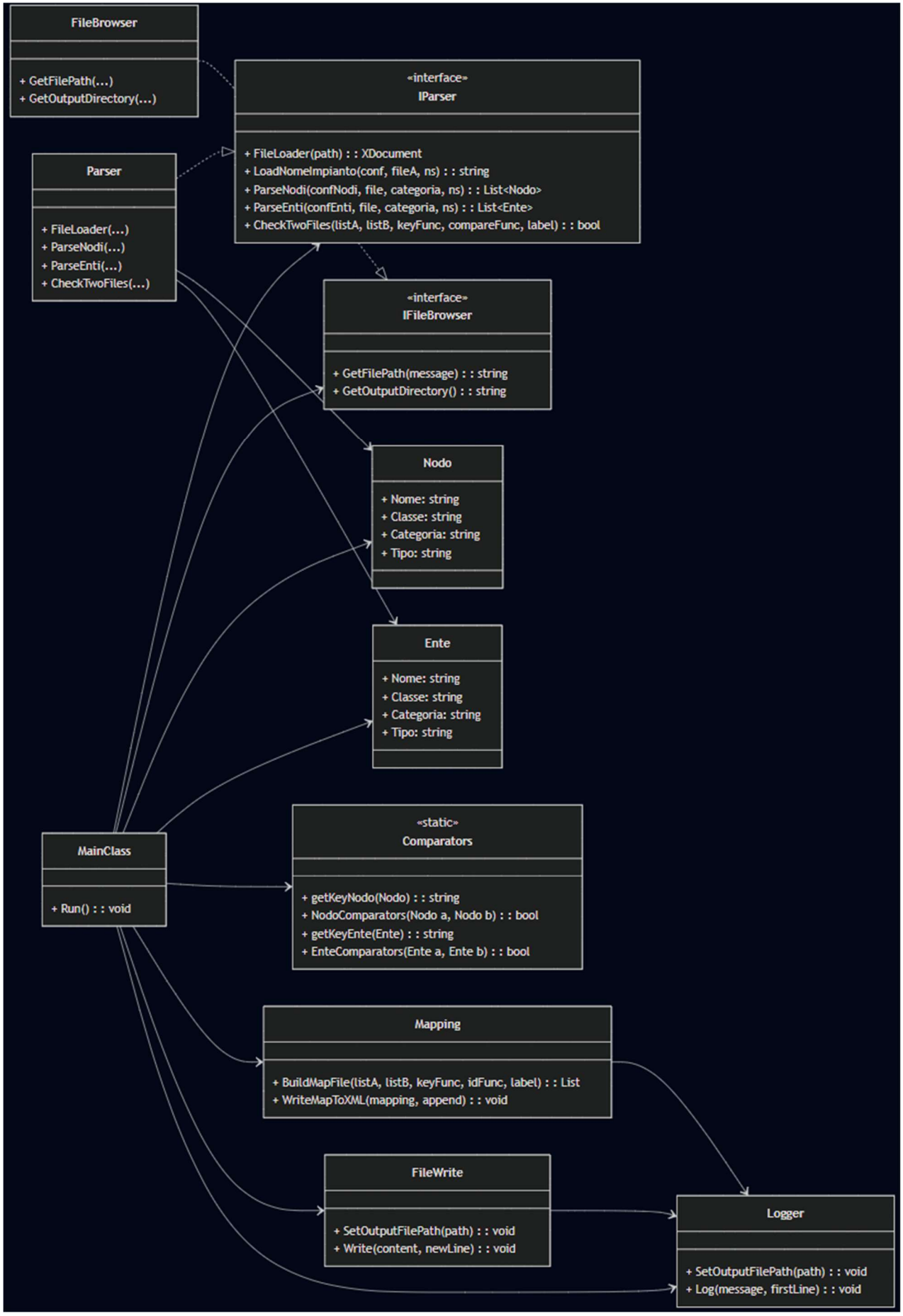

Figura 10: Class Diagram raffigurante l'architettura del SW.

4.2 Flusso Operativo del Software e Funzionamento del Mapping

Il flusso operativo del software è costituito da più fasi logiche ben distinte.

La gestione complessiva del processo è affidata alla *MainClass,* che coordina il funzionamento dei vari moduli e permette quindi, la cooperazione e lo scambio dati tra essi. La prima fase del flusso operativo è la fase di *selezione dati e inizializzazione:*

1. **Fase di Selezione e Inizializzazione:**

Il software inizia il proprio flusso operativo chiedendo all'utente di selezionare i file XML in input e la cartella di destinazione dei risultati.

Questa operazione è gestita dal modulo FileBrowser, che utilizza le finestre di dialogo di Windows per selezionare file e directory.

```
public string GetFilePath(string dialogTitle)
{
    //Abilita la visualizzazione delle finestre di dialogo
    Application.EnableVisualStyles();
    using (var openFileDialog = new OpenFileDialog())
    {
        openFileDialog.Title = dialogTitle;
        openFileDialog.Filter = "XML files (*.xml)|*.xml|All files (*.*)|*.*";
        if (openFileDialog.ShowDialog() == DialogResult.OK)
        {
            return openFileDialog.FileName;
        }
        else
        {
            return string.Empty;
        }
    };
}
```

Snippet che mostra il funzionamento del metodo di file browser per ottenere i percorsi

Parallelamente, i moduli Logger, FileWrite e Mapping impostano i rispettivi percorsi di output, sincronizzandosi con la scelta dell'utente.

Dopo questa prima fase il flusso operativo si incentra sulla seconda fase, ovvero il *parsing* effettivo *e la costruzione delle strutture dati*.

2. **Fase di Parsing e costruzione delle strutture dati:**

in questa fase entra in gioco il modulo *Parser,* che si occupa di caricare i 3 file, attraverso il metodo *FileLoader,* successivamente vengono estratti i nomi dell'impianto da tutti i file, e prodotta una prima lista di Nodi e di Enti, caricati direttamente dal file di configurazione, tramite metodi appositi.

```
public List<Nodo> parseConfFIleNodes(XDocument conf_file, XNamespace? ns = null)
 {
     var confNodi = new List<Nodo>();
     foreach (var confNodeElement in conf_file.Descendants(ns + "nodo"))
     {
         var confNode = new Nodo(confNodeElement, ns);
         confNodi.Add(confNode);
     }
     return confNodi;
 }
```

Snippet del metodo per ricavare i nodi dal file di configurazione. La lista di nodi contenuta nel file viene convertita in una lista di oggetti Nodo, questo tipo è definito all'interno del file XML_Element.cs

Dopo aver caricato i Nodi e gli enti del file di configurazione, ed aver ottenuto le liste corrispondenti, viene fatto il parsing anche di enti e nodi degli altri due file XML, il tutto basandosi sul risultato del parsing precedente attuato sul file di configurazione.
Ovviamente per questa operazione vengono utilizzati metodi differenti. Tutte le operazioni sull'estrazione degli oggetti interessati vengono registrati, e scritti all'interno del file di log tramite il modulo *Logger*.
Quando termina la ricerca si passa alla fase successiva, ovvero la fase di *confronto dati.*

3. Confronto dei File

Questa parte sfrutta il metodo generico *CheckTwoFiles<T>()*, che permette di verificare differenze tra liste di elementi di qualsiasi tipo (Nodo, Ente, ecc.), sfruttando funzioni di confronto specifiche definite nel modulo *Comparators*.

```
public bool CheckTwoFiles<T>(
    List<T> elemFile1,
    List<T> elemFile2,
    Func<T, string> getKey,
    List<Func<(T a, T b), (string fieldName, object val1, object val2)>>
     comparators,
    string entityName = "Elemento"
    )
{
    if(elemFile1 == null || elemFile2 == null)
    {
        throw new ArgumentNullException("Le liste di elementi non possono
        essere null.");
    }

    bool checkError = false;
    var elemFile2Dict = elemFile2.ToDictionary(getKey);
    string line = string.Empty;

    foreach (var elem in elemFile1)
    {
        var elemName = getKey(elem);
        if (elemFile2Dict.TryGetValue(elemName, out var elem2))
        {
            foreach (var comparer in comparators)
            {
                var (fieldName, val1, val2) = comparer((elem, elem2));
                if (!Equals(val1, val2))
                {
                    line = $"Discrepanza trovata: {entityName} {elemName}
                    {fieldName} diversa tra i due file --> File1: {val1} -
                    File2: {val2}";
                    Console.WriteLine(line);
                    FileWrite.Write(line, true);
                    checkError = true;
                }
            }
        }
        else
        {
            FileWrite.WriteMissingElement("secondo", entityName, elemName);
            Logger.Log(line, false);
            checkError = true;
        }
```

```
            }
            var nomiFile1 = elemFile1.Select(getKey).ToList();
            foreach (var elem2 in elemFile2)
            {
                var elemName = getKey(elem2);
                if (!nomiFile1.Contains(elemName))
                {
                    FileWrite.WriteMissingElement("primo", entityName, elemName);
                    Logger.Log(line, false);
                    checkError = true;
                }
            }
            return checkError;
        }
    }
```

Snippet del metodo CheckTwoFiles

Questo approccio generico consente di isolare la logica di confronto dai dettagli implementativi delle classi, migliorando la riusabilità e la manutenibilità del codice, e riducendone la duplicazione. Dopo aver effettuato i confronti si passa alla fase finale ovvero la *mappatura.*

4. Generazione del file di mappatura

In questa fase il lavoro viene svolto quasi interamente dal modulo di *Mapping,* che riceve in input le liste di oggetti confrontati e produce un file di mapping XML, nel quale vengono associati i nodi e gli enti equivalenti tra i due impianti.

```
public static void WriteMapToXML(
    List<(string entityName, string nome, double? idA, double? idB)> mapping,
    bool append
    )
{
    XDocument doc;
    XElement root;

    if (File.Exists(outputPath) && append)
    {
        doc = XDocument.Load(outputPath);
        root = doc.Root ?? new XElement("mapping");
        if (doc.Root == null)
```

```
        {
            doc.Add(root);
        }
    }
    else
    {
        root = new XElement("mapping");
        doc = new XDocument(new XDeclaration("1.0", "", ""), root);
    }

    foreach (var m in mapping)
    {
        var entityTag = m.entityName == "Nodo" ? "nodo" : "ente";
        var mapElement = new XElement(entityTag,
            new XAttribute("nome", m.nome),
            new XAttribute("id-A", m.idA?.ToString() ?? string.Empty),
            new XAttribute("id-B", m.idB?.ToString() ?? string.Empty)
        );
        root.Add(mapElement);
    }

    doc.Save(outputPath);
}
```

Snippet del modulo per costruire il mapping

Questo file consente quindi di ricostruire in modo univoco la relazione tra i due sistemi, rappresentando il punto di partenza per successive analisi o migrazioni di dati.

Infine, l'intero processo è accompagnato da una registrazione dettagliata delle operazioni tramite il modulo *Logger*, che annota ogni fase significativa — dal caricamento dei file, al parsing, fino all'esito del confronto, registrando ogni fase con questo timestamp:

```
writer.WriteLine($"{DateTime.Now:yyyy-MM-dd HH:mm:ss:fff} - {message}");
```

Questo consente una tracciabilità completa, utile per attività di debugging o audit del processo. La combinazione tra modularità, generics e logging strutturato rende il software flessibile, facilmente estendibile e adatto all'utilizzo in contesti aziendali in cui è richiesta affidabilità e chiarezza operativa.

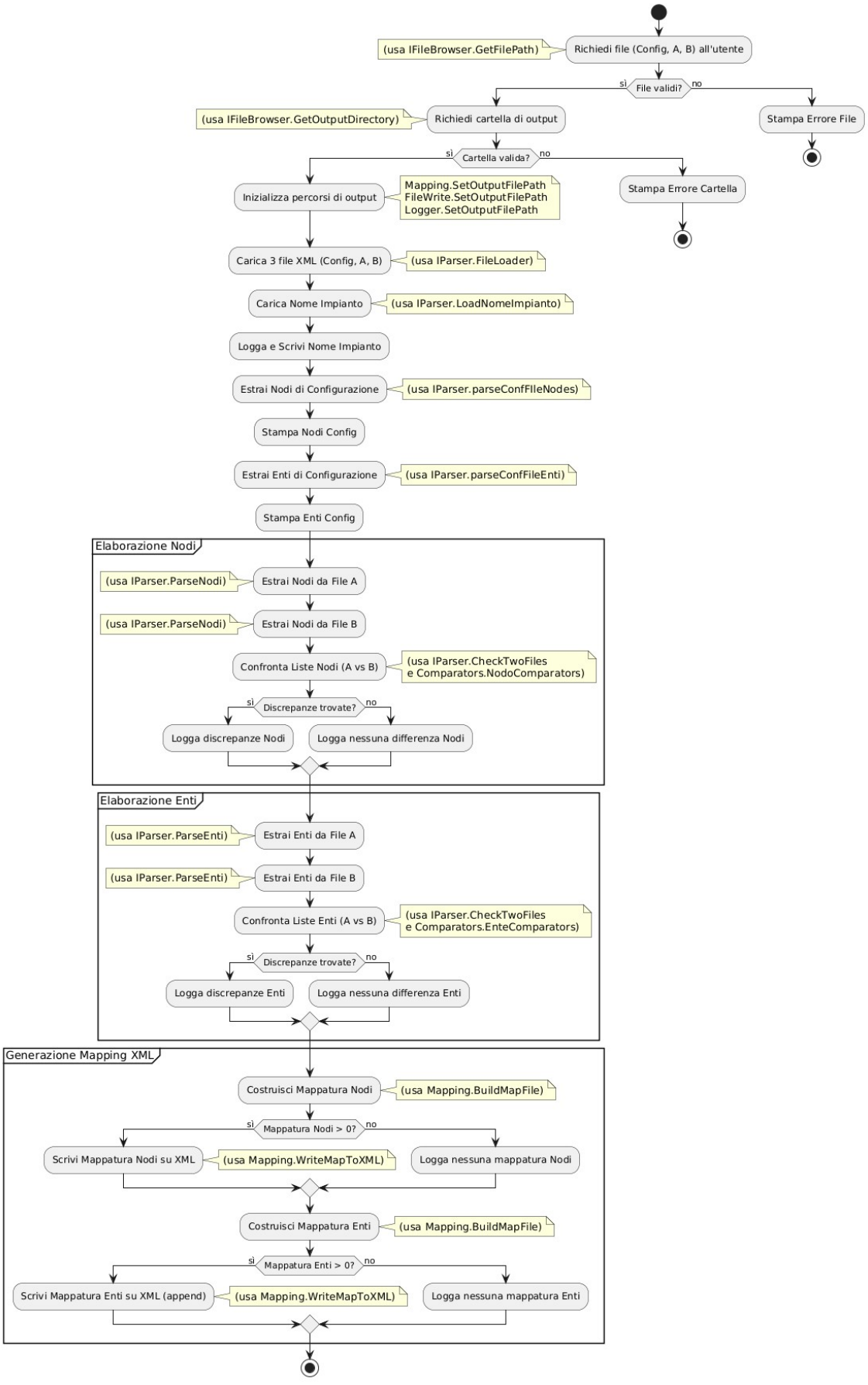

Figura 11: Activity Diagram che mostra il flusso operativo del programma

4.3 Produzione dei Risultati

Nel paragrafo precedente è stato illustrato il flusso operativo completo del programma; si analizzano ora nel dettaglio i risultati prodotti al termine dell'esecuzione.

Come già descritto, la generazione del file di mappatura è interamente delegata al modulo *Mapping*, il quale si occupa di associare in maniera strutturata nodi ed enti appartenenti ai due impianti confrontati. Per quanto riguarda invece gli altri due output principali — **Report** e **Log** — il lavoro è completamente gestito dai moduli *FileWrite* e *Logger*. Questi componenti sono responsabili della scrittura continua e incrementale dei rispettivi file durante tutte le fasi del processo, registrando informazioni rilevanti ogni volta che viene eseguita un'operazione significativa.

Modulo Logger: registrazione cronologica delle operazioni

Il modulo Logger è progettato per avere responsabilità unica: scrivere messaggi di log con timestamp, senza conoscere la logica applicativa. Si occupa di annotare ogni fase importante dell'elaborazione, includendo caricamento dei file, risultati intermedi, eventuali errori o discrepanze.

```
public static class Logger
{
    private static string logPath = "";

    public static void SetOutputFilePath(string directory)
    {
        logPath = Path.Combine(directory, "log.txt");
    }
    public static void Log(string message, bool firstLog)
    {
        using (StreamWriter writer = new StreamWriter(logPath, append: !firstLog))
        {
            writer.WriteLine($"{DateTime.Now:yyyy-MM-dd HH:mm:ss:fff} - {message}");
        }
    }
}
```

Snippet Contenente il funzionamento del logger

Il flag *firstLog* permette di creare un nuovo log all'inizio dell'esecuzione e poi appendere per tutte le operazioni successive. La figura 12 mostra un esempio di contenuto del *log.*

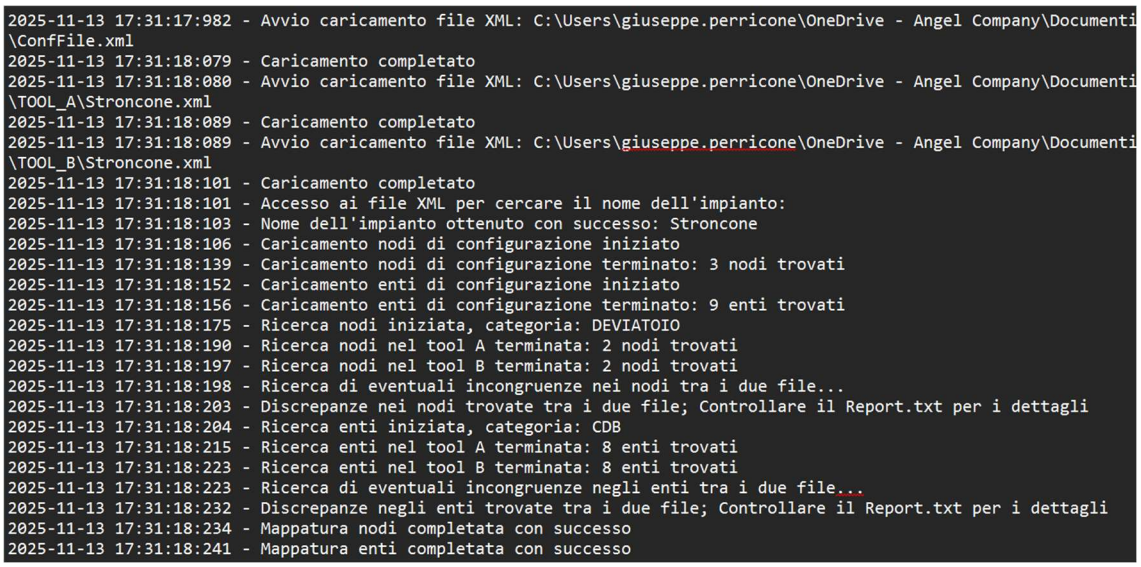

Figura 12: Esempio di contenuto del Log

Modulo FileWrite: Produzione del Report

Il modulo FileWrite invece gestisce esclusivamente la creazione del documento *Report.txt*, il quale raccoglie: l'elenco dei nodi ed enti caricati del file di configurazione, l'elenco dei nodi ed enti trovati nei rispettivi file e mostra le eventuali differenze individuate durante il confronto, compresi gli elementi mancanti.

```
public static class FileWrite
{
    private static string outputFile = "";
    public static void SetOutputFilePath(string path)
    {
        outputFile = Path.Combine(path, "Report.txt");
    }
    public static void Write(string content, bool append)
    {
        using (StreamWriter writer = new StreamWriter(outputFile, append: append))
        {
            writer.WriteLine(content);
```

```
        }
    }

    public static void WriteMissingElement(string fileName, string entityName,
string elemName)
    {
        var line = $"{entityName} {elemName} non trovato nel {fileName} file.";
        Console.WriteLine(line);
        FileWrite.Write(line, true);
    }
}
```

Snippet che mostra il funzionamento del FileWriter

Grazie al parametro append, il sistema decide se creare il file o aggiungere nuovi contenuti. Nella figura 13 viene mostrato un esempio di contenuto del report.

```
Nodi di configurazione:
CATEGORIA  TIPO      NOME  CLASSE
DEVIATOIO  SEMPLICE  02    PESANTE
DEVIATOIO  SEMPLICE  01    PESANTE
DEVIATOIO  SEMPLICE  85    PESANTE

Enti di configurazione:
CATEGORIA  TIPO      NOME  CLASSE
CDB        STAZIONE  120   SENZA ENTI
CDB        STAZIONE  121   SENZA ENTI
CDB        STAZIONE  122   CON ENTI
CDB        STAZIONE  112   CON ENTI
CDB        STAZIONE  301   SENZA ENTI
CDB        STAZIONE  302   SENZA ENTI
CDB        STAZIONE  111   SENZA ENTI
CDB        STAZIONE  110   SENZA ENTI
CDB        STAZIONE  756   SENZA ENTI

--------------------------------------------------------------------------------

Ricerca di nodi di tipo: DEVIATOIO...

Risultati da TOOL_A:
 - NOME       CLASSE             ID          X          Y
 - 02         PESANTE           882     286401      99022
 - 01         PESANTE           944     249381      99022
Nodo 85 non trovato nel file

Risultati da TOOL_B:
 - NOME       CLASSE             ID          X          Y
 - 02         PESANTE           882     288401      99022
 - 01         PESANTE           944     249381      99022
Nodo 85 non trovato nel file

Ricerca di eventuali incongruenze nei nodi tra i due file:
Discrepanza trovata: Nodo 02 X diversa tra i due file --> File1: 286401 - File2: 288401
```

Figura 13: Esempio di Report

Le scritture su report e log avvengono in punti diversi del programma, a seconda della natura dell'informazione da registrare.

Un esempio significativo è riportato di seguito:

```
List<Ente> entiB = _parser.ParseEnti(confEnti, fileB, categoriaEnteInteressata, ns);
Logger.Log("Ricerca enti nel tool B terminata: " + entiB.Count + " enti trovati",
firstLog);


Logger.Log("Ricerca di eventuali incongruenze negli enti tra i due file...",
firstLog);
Console.WriteLine($"\nRicerca di eventuali incongruenze negli enti tra i due file:");
FileWrite.Write($"\nRicerca di eventuali incongruenze negli enti tra i due file:",
firstline);
```

Snippet che mostra una chiamata di Logger e FileWriter

A differenza della mappatura, che è generata in un punto preciso e tramite un modulo dedicato, la creazione di Report e Log non è centralizzata.

Le scritture avvengono infatti durante l'intero flusso di esecuzione, ciascuna nel punto in cui l'evento diventa rilevante. I dettagli sul funzionamento della mappatura sono stati mostrati nel paragrafo precedente, la figura 14 tuttavia mostra un esempio di contenuto del mapping

```
▼<mapping>
   <nodo nome="02" id-A="882" id-B="882"/>
   <nodo nome="01" id-A="944" id-B="944"/>
   <ente nome="120" id-A="2025" id-B="31806"/>
   <ente nome="121" id-A="2882" id-B="2882"/>
   <ente nome="122" id-A="2986" id-B="2986"/>
   <ente nome="112" id-A="3476" id-B="3476"/>
   <ente nome="301" id-A="3781" id-B="3781"/>
   <ente nome="302" id-A="3794" id-B="3794"/>
   <ente nome="111" id-A="3963" id-B="3963"/>
   <ente nome="110" id-A="4046" id-B="4046"/>
 </mapping>
```

Figura 14: Esempio di contenuto del mapping

4.4 Descrizione dei Casi di Unit Test

Come anticipato nel Capitolo 3, il software è stato accompagnato da un'ampia suite di test che ha permesso di validare tutte le funzionalità principali: caricamento dei file, parsing, confronto dei dati, generazione dei file di output (Report, Log, Mapping) e

gestione degli scenari anomali.

L'intera test suite è stata sviluppata utilizzando il framework NUnit per la struttura dei test e Moq per l'isolamento delle dipendenze, in modo da testare ogni modulo senza effetti indesiderati derivanti da I/O reale.

La suite di test è organizzata in tre grandi blocchi:

- **ParserTests:** test sulle funzionalità di parsing XML, caricamento file, estrazione nodi/enti e logiche di filtro. L'obiettivo è quello di validare l'integrità dei dati letti dai 3 file e la corretta gestione delle configurazioni
- **ComparatorTests:** test sul metodo generico *CheckTwoFiles*<T>() e sui comparatori specifici per Nodo e Ente. L'obiettivo è quello di garantire che differenze, anomalie e corrispondenze siano rilevate correttamente.
- **IOMappingHelperTests:** test sui moduli di output (Logger, FileWrite, Mapping). L'obiettivo è quello di assicurare la correttezza dei file generati e del comportamento dei servizi I/O.

In Figura 15 è riportata una tabella che sintetizza tutti i test implementati:

ID	Funzionalità	Priorità	Metodo Testato	Descrizione del Caso	Risultato Atteso
P01	Parsing	Alta	FileLoader	Fornire un file XML valido.	Caricamento riuscito (XDocument non nullo).
P02	Parsing	Alta	Flusso File/Caricamento	(Tramite Mocking) Verificare che il percorso del file selezionato dall'utente venga passato correttamente a parser.FileLoader().	La funzione parser.FileLoader() è chiamata con la stringa c
P03	Parsing	Media	LoadNomeImpianto	Verificare l'estrazione del nome dell'impianto da un file XML.	Stringa con il Nome Impianto corretto.
P04	Parsing	Alta	parseConfFileNodes	Verificare che il numero e i dettagli dei Nodi estratti dal config.xml siano corretti.	Lista di Nodo con numero e dati attesi.
P05	Parsing	Media	ParseNodi	Verificare l'estrazione solo dei Nodi con categoria "DEVIATOIO" dal file Tool.	Lista contenente solo Nodi di categoria "DEVIATOIO".
C01	Confronto	Critica	CheckTwoFiles (Nodi)	Fornire due liste identiche di Nodi.	false (Nessuna discrepanza e nessun errore loggato).
C02	Confronto	Critica	CheckTwoFiles (Nodi)	Fornire lista A con un Nodo in più (mancante in B).	true (Discrepanza trovata).
C03	Confronto	Critica	CheckTwoFiles (Nodi)	Fornire Nodi corrispondenti per chiave, ma con le coordinate X/Y diverse.	true (Discrepanza trovata sul contenuto).
C04	Confronto	Critica	CheckTwoFiles (Enti)	Fornire due liste identiche di Enti.	false (Nessuna discrepanza e nessun errore loggato).
C05	Confronto	Critica	CheckTwoFiles (Enti)	Fornire lista B con un Ente in più (mancante in A).	true (Discrepanza trovata).
C06	Confronto	Alta	CheckTwoFiles (Enti)	Confronto tra due oggetti Enti che differiscono solo per l'attributo Id.	true (Il comparatore rileva la differenza).
I01	I/O & Report	Alta	FileWrite	Verificare che i dati vengano scritti nel file di report con formattazione e allineamento (es. l'header).	Testo scritto nel report correttamente formattato.
I02	I/O & Logger	Media	Logger	Verificare l'inizializzazione corretta del file di log (firstLog = true) e l'aggiunta successiva di messaggi.	File di log creato e contenente tutti i messaggi.
M01	Mapping	Alta	Mapping.BuildMapFile	Verificare che la lista di mapping contenga le corrette associazioni (ID A → ID B).	Lista di mapping con corrispondenze esatte.
M02	Mapping	Media	Mapping.WriteMapToXML	Verificare che, con appendXML = true, il mapping degli Enti venga aggiunto senza sovrascrivere il mapping dei Nodi.	File XML contenente sia il mapping Nodi che Enti.
H01	Helper	Bassa	PrintConfigNodes	Verificare la gestione di una lista di Nodi vuota.	Messaggio "(nessun nodo di configurazione)" stampato in

Figura 15: Tabella raffigurante la test suite per il progetto

A seguito dell’implementazione ed esecuzione di questa test suite, si può correttamente affermare che il livello di copertura funzionale del sistema risulta elevato e sufficiente a garantire affidabilità del software nelle condizioni operative reali.