Documentazione AutomaZ

Giulio Bovina – VR485736

David Zahariev – VR489246

Febbraio 2025

Introduzione

Questa documentazione ha lo scopo di guidare al corretto utilizzo del software e a introdurne le funzionalità. Si rivolge sia ad utenti alle prime armi che a utenti più esperti.

Glossario

- “Word Automata” o “Automa”: un tipo di macchina a stati finiti il quale determina l’accettazione o il rifiuto di una data parola.
- “Nodo” o “Stato”: uno degli stati che compongono l’automa.
- “Edge” o “Arco”: una delle transizioni da un nodo a un altro.

Il sistema si propone di fornire gli strumenti grafici per creare e modificare il proprio grafo e gli strumenti per l’inserzione di una parola sul quale l’automa verrà eseguito. Inoltre, fornisce un sistema di salvataggio dei grafi creati e di apertura dei grafi salvati.

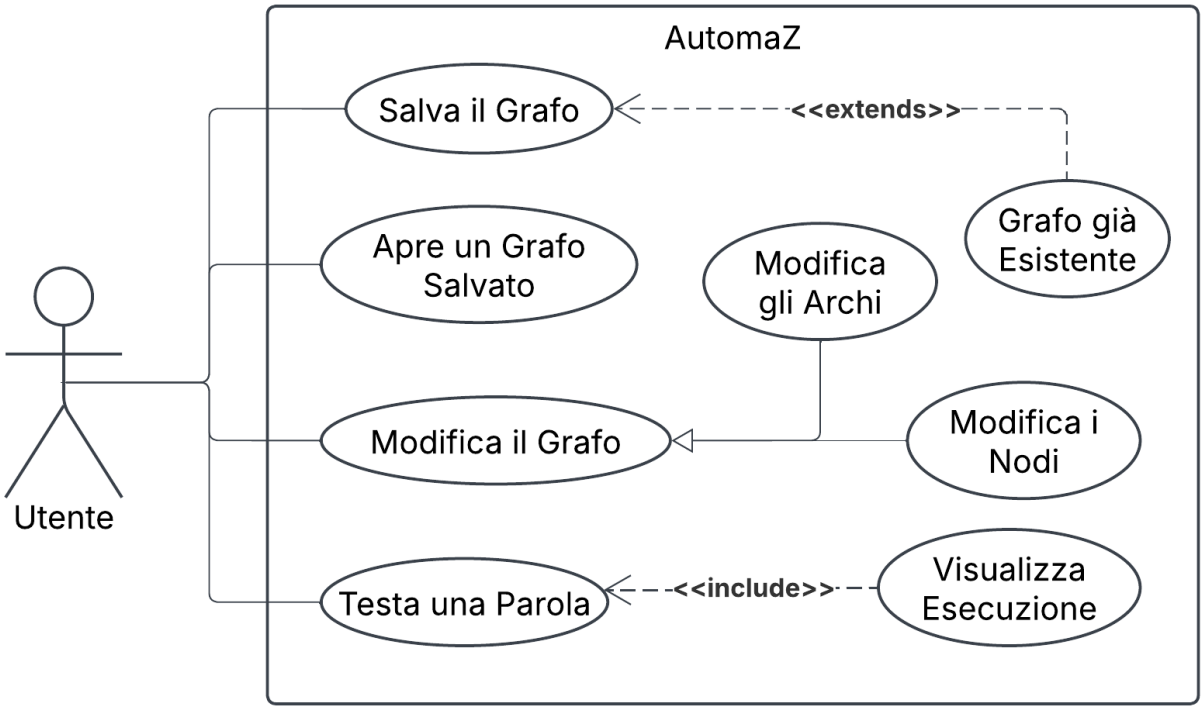


Diagramma dei Casi d’Uso

Spiegazione Automa

L'automa trattato dal software è nello specifico un DFSA ovvero un Deterministic Finite State Automa. Tale automa è definito e identificato dalla seguente quintupla:

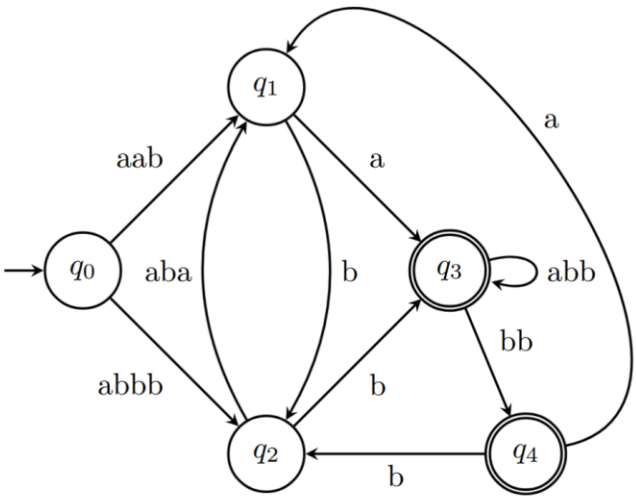
$$A =< Q, q_0, F, \Sigma, \delta >$$

dove:

- Q è l'insieme degli stati.
- $q_0 \in Q$ è lo stato iniziale.
- F è l'insieme degli stati finali.
- Σ è l'alfabeto, ovvero l'insieme dei simboli riconosciuti dall'automa.
- δ è l'insieme delle transizioni tra un nodo e un altro.

Un automa per essere eseguito deve prima di tutto essere definito in tutte e cinque queste componenti dopodiché deve essere fornita una parola, la quale verrà "consumata" dall'automa. La terminazione corretta di questo automa (chiamata anche l'accettazione della parola da parte dell'automa) significa che l'intera parola è stata consumata e che lo stato in analisi a fine esecuzione fa parte dell'insieme degli stati finali.

Di seguito un esempio di automa, rappresentato con un grafo direzionale, e tre casi di esecuzione:



aababbbb

q_0	q_1	q_3	q_4	q_2	q_3	
	aab	*a*	*bb*	*b*	*b*	✓

Caso 1

abbbaba

q_0	q_2	q_1	
	abbb	*aba*	✗

Caso 2

abbbabaaba

q_0	q_2	q_1	q_3	
	abbb	*aba*	*a*	~~*ba*~~ ✗

Caso 3

Requisiti

Durante la modifica del grafo l'utente è libero di fare ciò che preferisce. Tuttavia, prima della fase di esecuzione vengono effettuati dei controlli di correttezza del grafo creato.

Requisiti di Correttezza

1. **Stato Iniziale Univoco**: Non possono esistere più di due stati iniziali (o meno di uno).
2. **Transizioni Non Vuote**: Non possono esistere transizioni il cui valore non viene specificato o viene lasciato vuoto.
3. **Determinismo 1**: Nel caso in cui due transizioni abbiano lo stesso stato di origine esse non possono avere lo stesso valore.
4. **Determinismo 2**: Nel caso in cui due transizioni hanno lo stesso stato di origine e sono entrambe accettabili, deve sempre essere scelta la transizione dal valore più lungo.

Inoltre, dalle specifiche sono emersi anche i seguenti requisiti:

Requisiti Funzionali

1. **Rappresentazione Grafica dell'Automa**: Il grafo dell'automa viene rappresentato e modificato graficamente. L'utente è in grado di inserire, rimuovere e modificare stati e transizioni.
2. **Inserimento Parola ed Esecuzione**: Attraverso un campo di testo l'utente può inserire la parola sulla quale verrà eseguito l'automa.
3. **Visualizzazione del Processo**: Il software mostra il risultato di accettazione o rifiuto dell'esecuzione. Inoltre, viene mostrato in ordine di successione i nodi da cui l'automa passa e le transizioni prese tra un nodo e un altro. Nel caso in cui non tutta la parola venga consumata, si mostra la sotto parola restante.

Scelte Progettuali

Rappresentazione Nodo Iniziale

Per rendere più immediato il riconoscimento del nodo iniziale abbiamo deciso di colorarlo di giallo invece di rappresentarlo con una transizione senza nodo di origine.

Sovrapposizione Nodo Iniziale e Nodo Finale

A seguito di test con svariati tipi di grafi siamo giunti alla decisione di rendere possibile l'esistenza di un nodo che è sia iniziale che finale. Abbiamo incluso questa decisione nelle scelte progettuali in quanto:

1. Non è un requisito specificatamente richiesto.
2. Si tratta di un problema risolvibile attraverso un nodo ausiliario con una sola transizione verso il nodo finale. Questa soluzione però non è del tutto accettabile in quanto servirebbe definire una transizione vuota, specificatamente vietata dai requisiti. Pertanto, bisognerebbe aggiungere all'inizio di ogni parola da testare un prefisso predeterminato. La scelta evita tale bisogno

Rappresentazione Nome dei Nodi e Valore delle Transizioni

Per rendere la rinomina degli stati e la modifica dei valori delle transizioni meno impegnativa abbiamo deciso di rappresentare la lista intera di stati e transizioni in una colonna dedicata a tale scopo.

In questa colonna, effettuare il doppio click su uno stato o una transizione dà la possibilità di modificare rispettivamente il nome o il valore.

Inoltre, al passaggio del cursore sopra un elemento in questa colonna va ad evidenziare l'oggetto a cui esso si riferisce nella rappresentazione centrale del grafo.

Memorizzazione del Grafo

Nel software è stato implementato il salvataggio del grafo creato dall'utente in memoria come anche l'apertura dei grafi salvati precedentemente. Anche in questo caso tale scelta non faceva parte dei requisiti, tuttavia, riteniamo sia una parte importante del software in quanto alcuni grafi possono diventare molto complessi.

Casi d’Uso

Modifica del Grafo

La modifica del grafo può essere di tipo funzionale oppure di tipo esclusivamente grafica. Entrambi i tipi di modifiche sono suddivisi in due categorie, ovvero in base a quale tipo di oggetto viene modificato. Che esso sia un Nodo o un Arco. Più avanti nella documentazione se ne entrerà in dettaglio.

Test di una Parola

Questa attività prevede l’inserimento da parte dell’utente di una parola in un campo di testo. Nel momento in cui il pulsante di avvio viene premuto vengono effettuati i controlli necessari e se il grafo è valido, il sistema procede ad eseguire l’automa.
Da questa attività dipende quella di visualizzazione dell’esecuzione ovvero del risultato e del cammino preso, come anche di un’eventuale parola rimanente in caso di mancata terminazione.

Salvataggio del Grafo

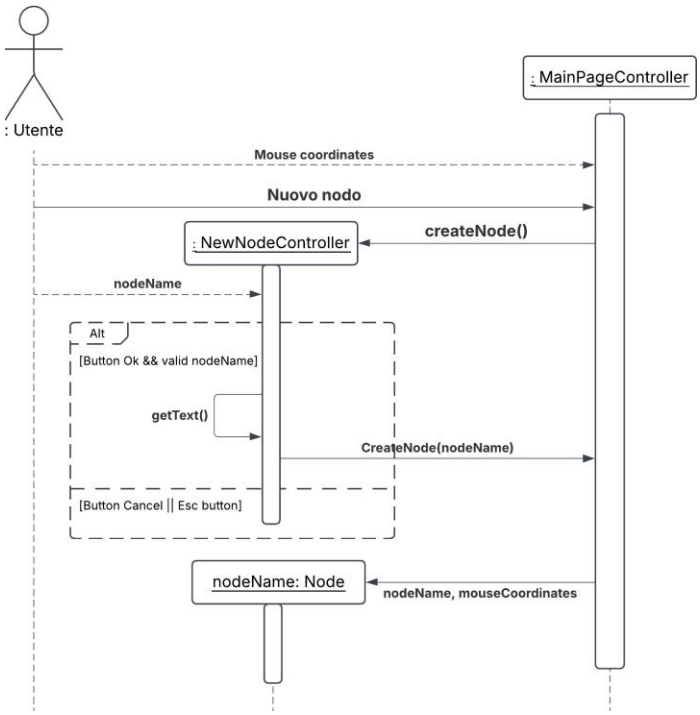
I grafi modificati possono essere salvati sotto forma di file di testo per poter essere utilizzati più tardi. Per fare ciò, il sistema chiede all’utente il nome del file in cui il grafo verrà salvato e, nel caso in cui esista già un file con il nome inserito, verrà notificato l’utente e richiesto di cambiare nome.

Apertura di un Grafo

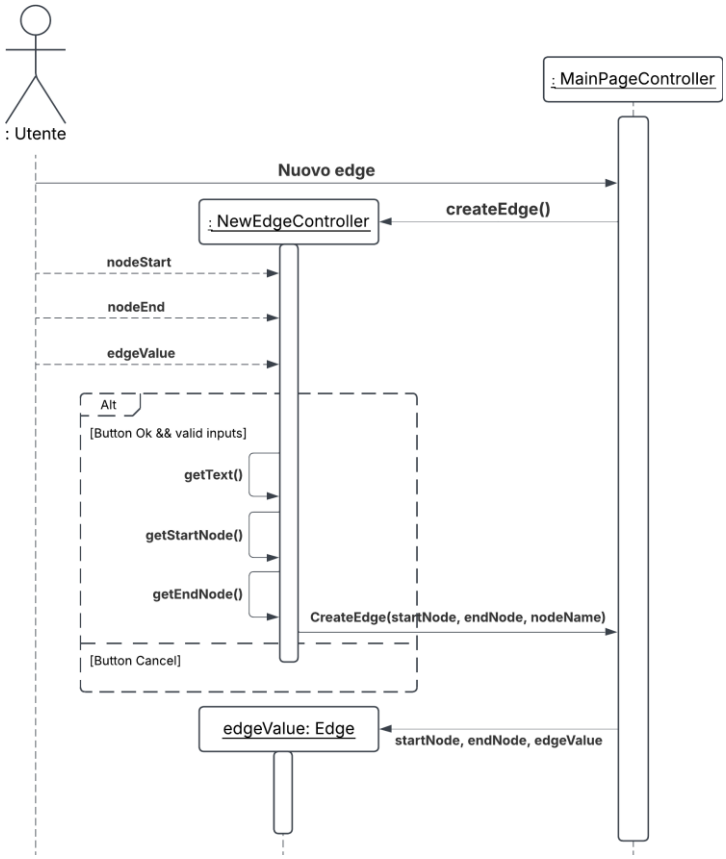
I grafi salvati in precedenza possono essere aperti. Per farlo viene proposta all’utente una lista di file già esistenti che il sistema ha trovato memorizzati. Per questo motivo non è possibile che l’utente causi errori come nel salvataggio del grafo.

Si possono trovare di seguito i diagrammi di sequenza dei casi d’uso appena spiegati.

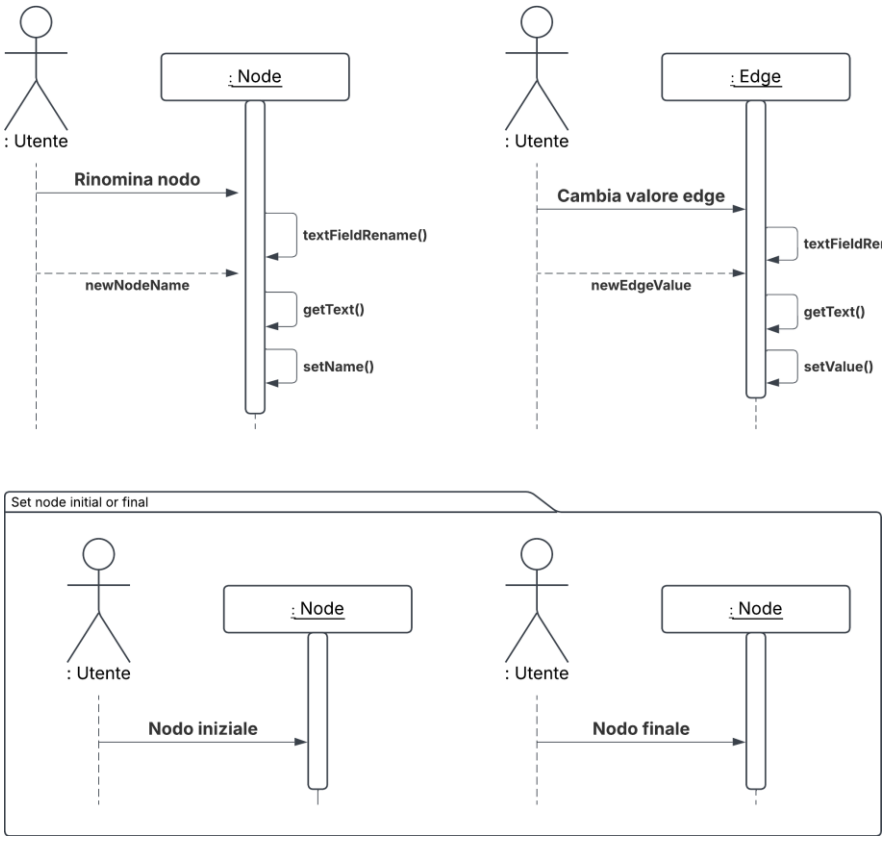
Diagrammi di Sequenza



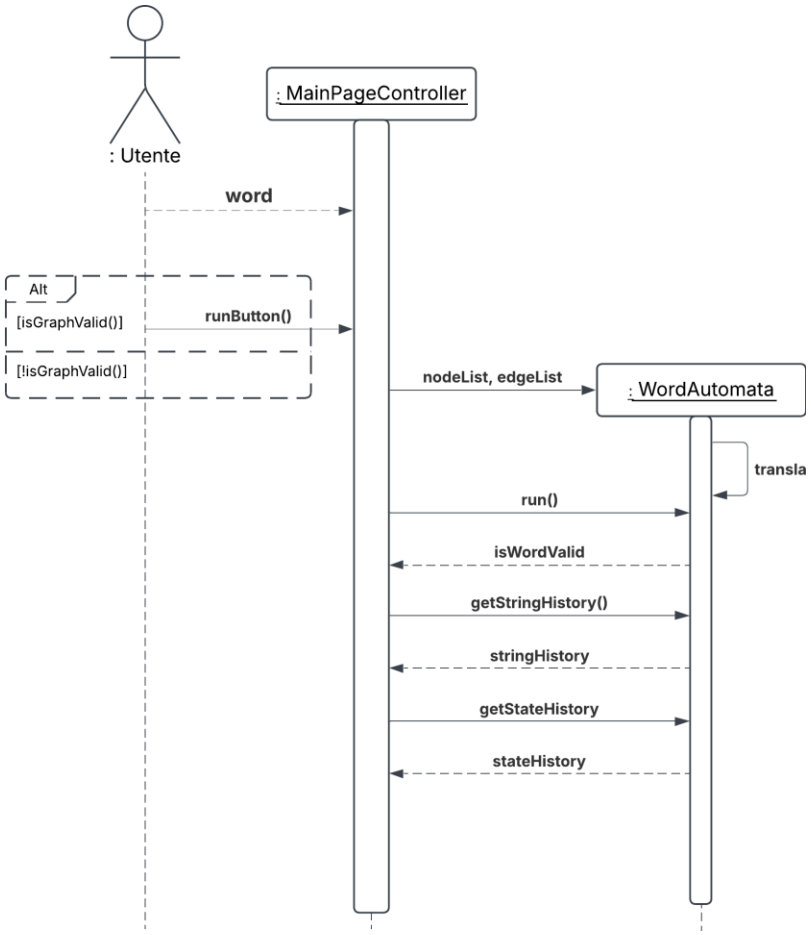
Creazione Nodo



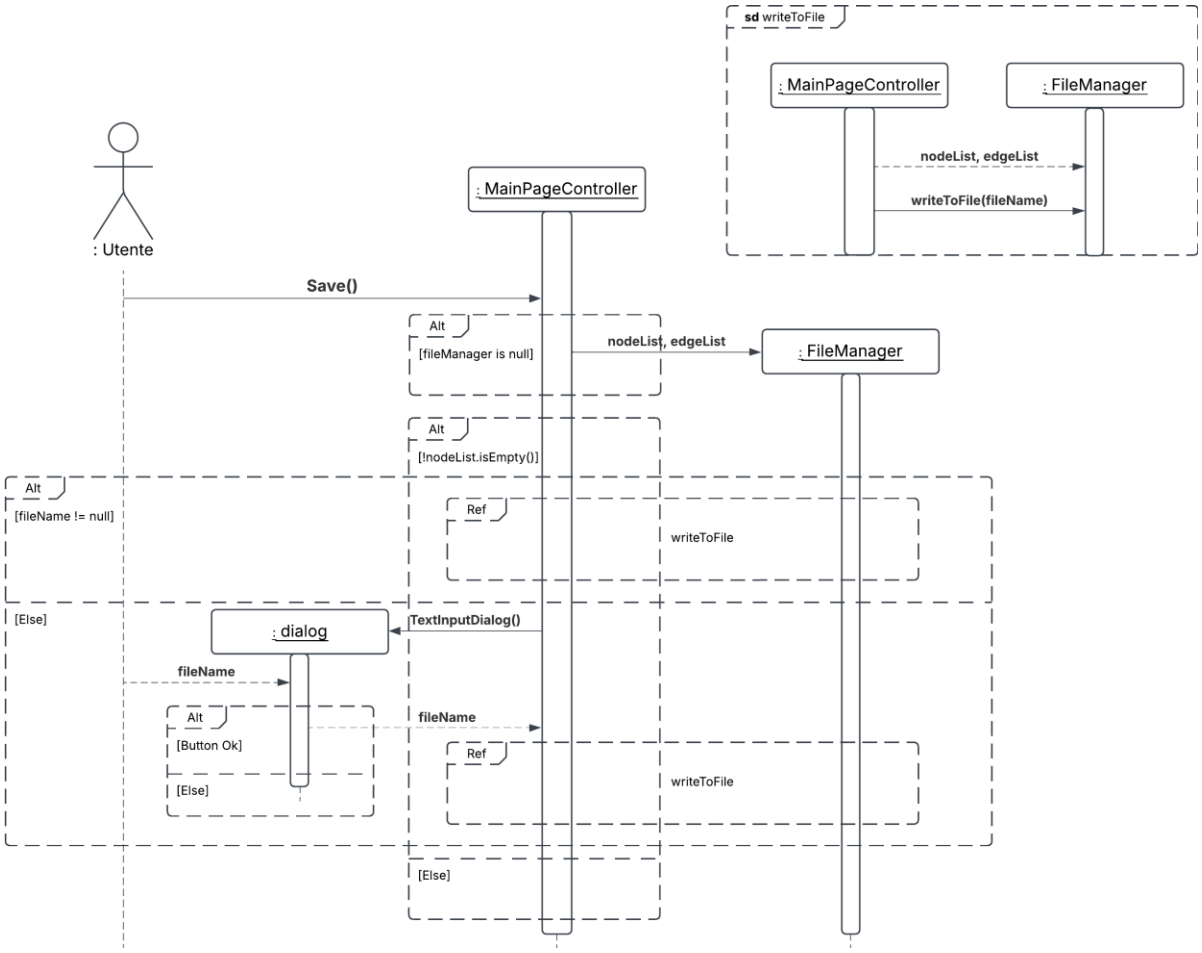
Creazione Edge



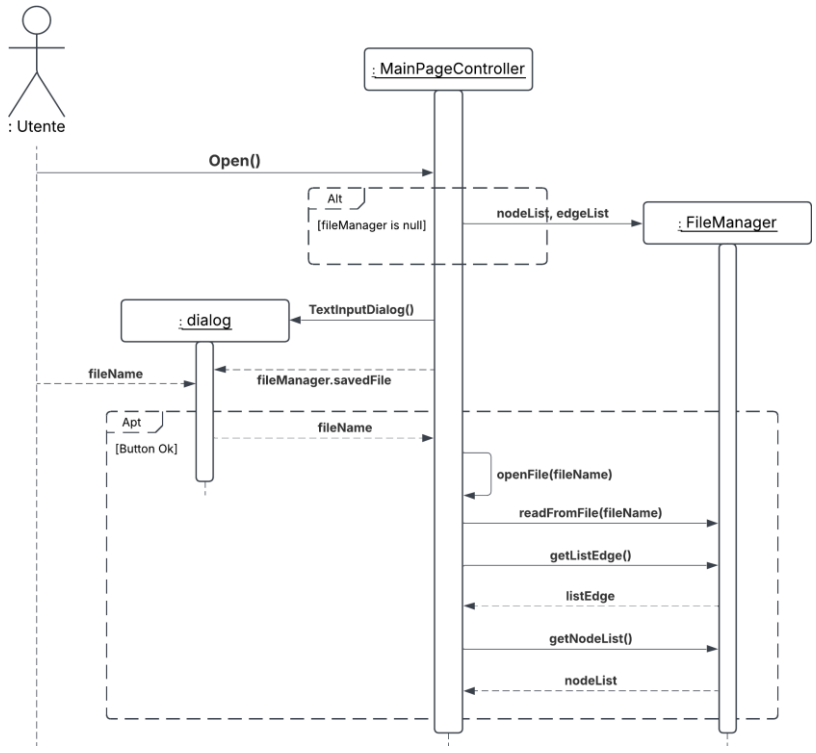
Modifica di Nodi e di Edge



Esecuzione Automa



Salvataggio Grafo



Apertura Grafo

Diagramma delle Classi

Nel seguente diagramma, per mantenere la visibilità, sono stati rimossi i metodi di tipo privato.

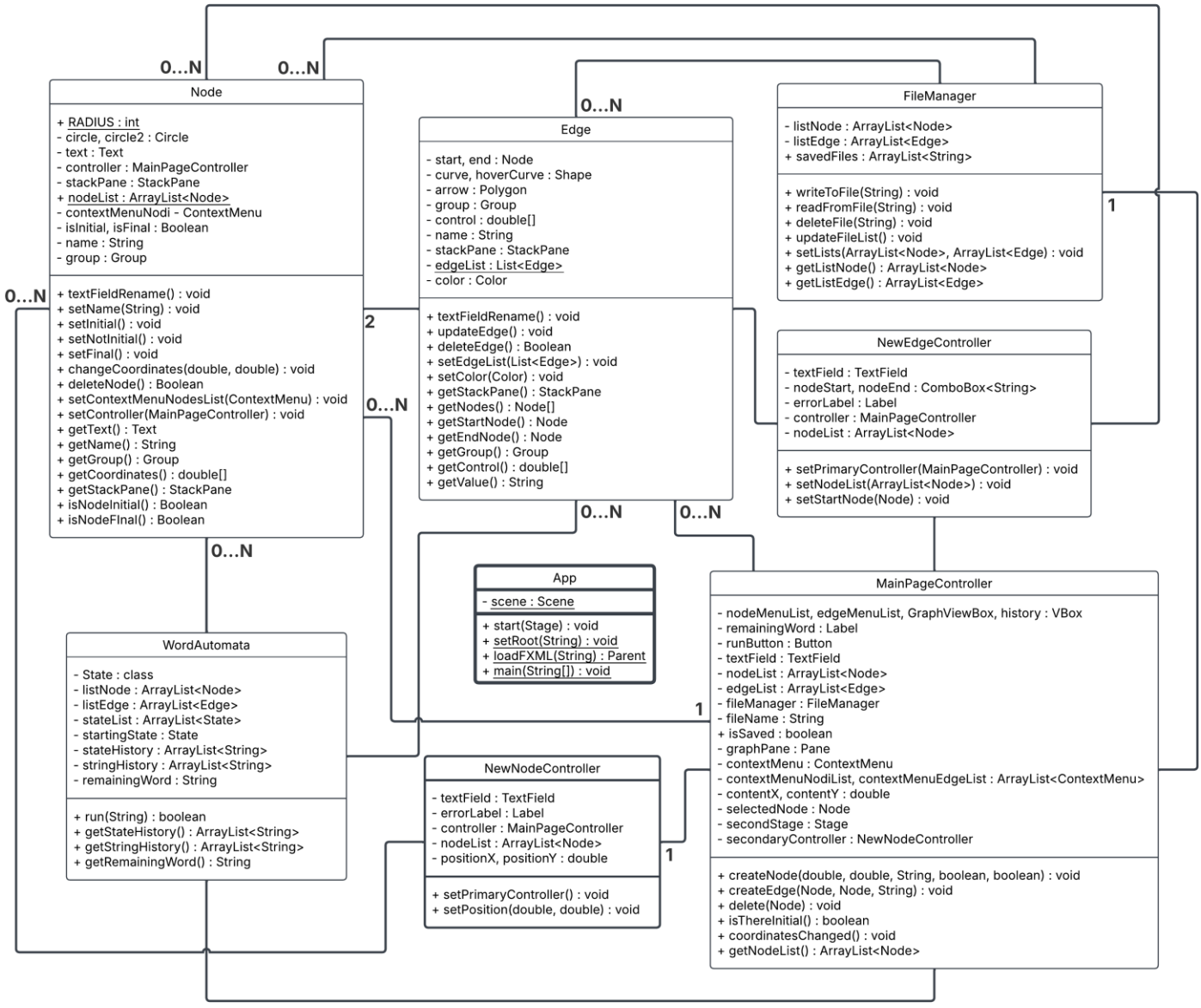
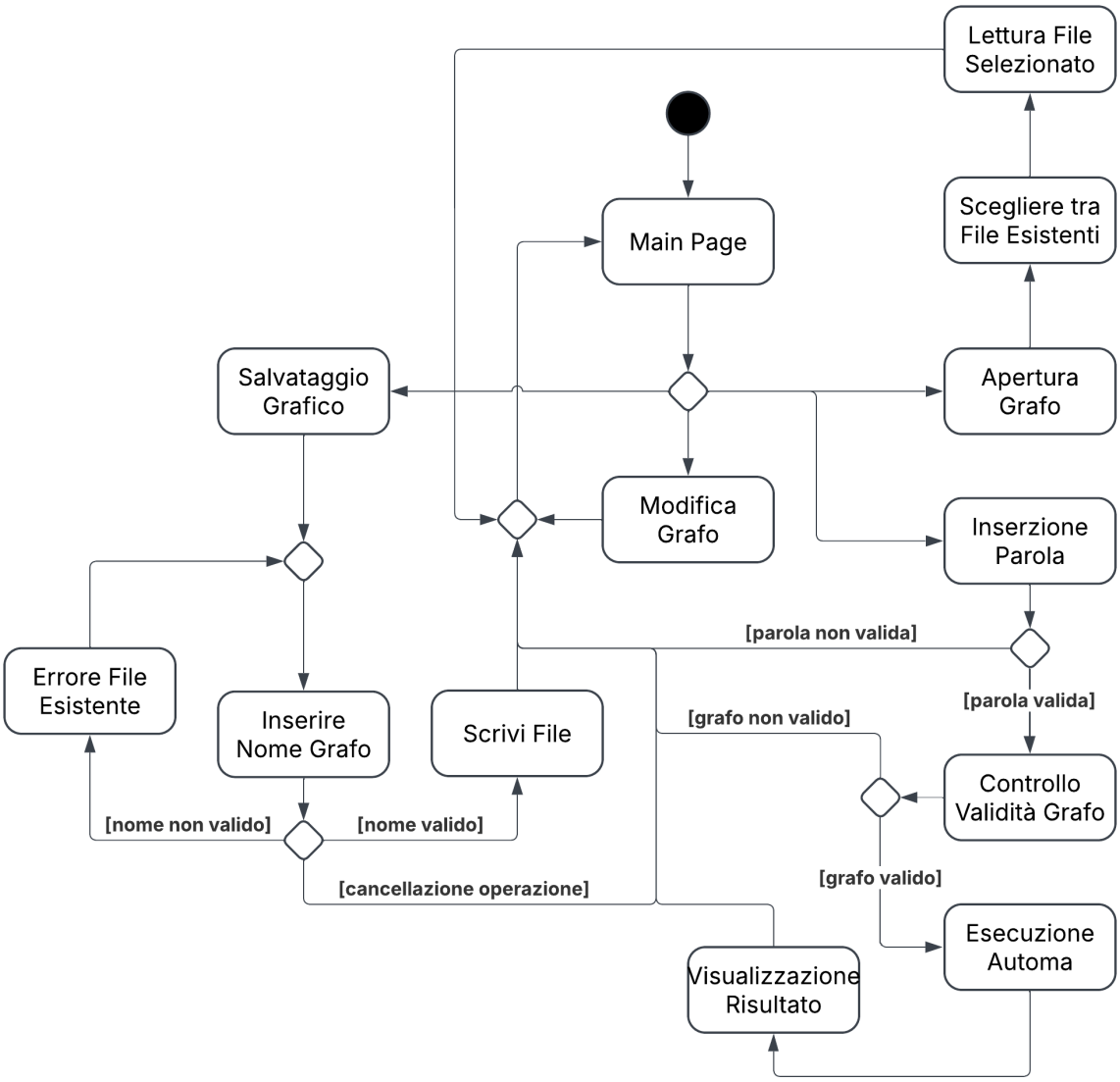


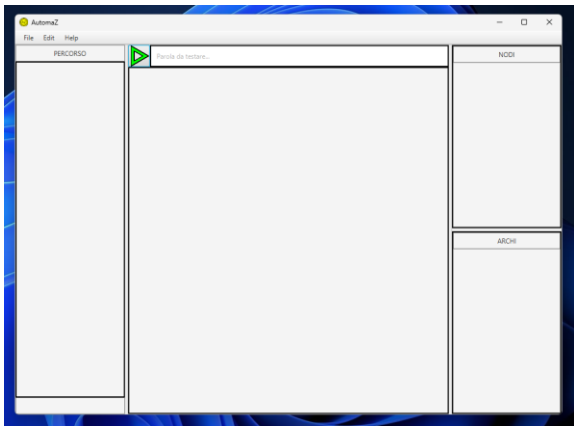
Diagramma delle Classi

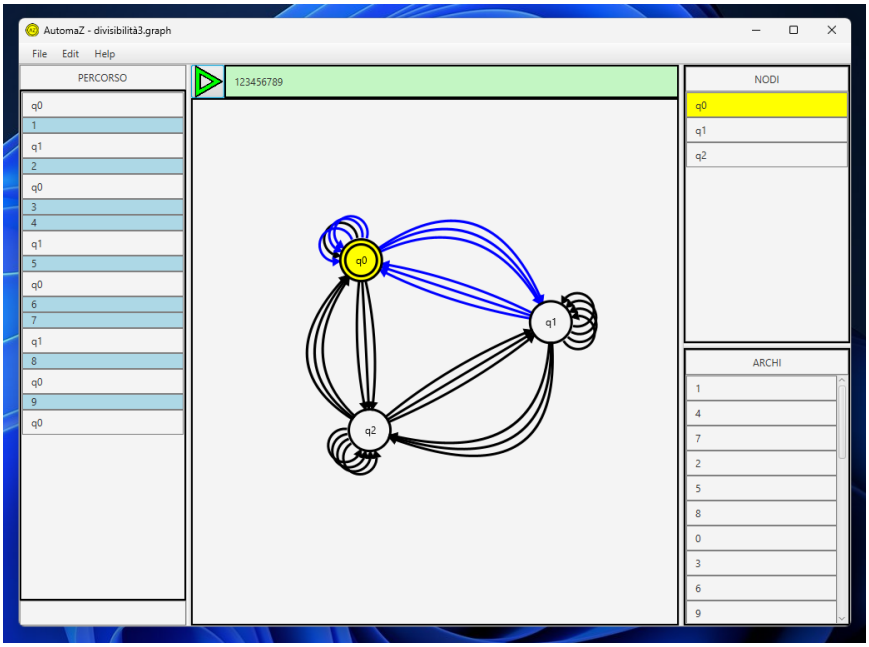
Diagramma delle Attività



Interfaccia Utente

Il sistema si presenta con una disposizione semplice ed intuitiva.
Viene suddiviso in tre colonne principali, ovvero (da sinistra a destra) percorso effettuato e parola rimanente, grafo e liste di nodi e archi, di cui solo la prima serve esclusivamente a mostrare informazioni. Le altre due hanno anche funzioni di interazione con l'utente.
Inoltre la finestra è ridimensionabile.

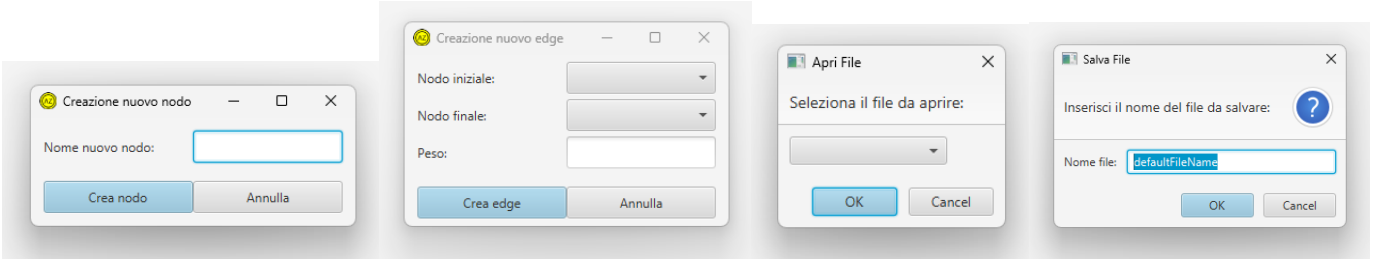




Guardiamo subito un esempio.
L'automa rappresentato qui a sinistra serve per determinare se un numero inserito è divisibile per 3.
Possiamo vedere come l'inserzione della parola "123456789" venga accettata dall'automa e anche il percorso preso durante l'elaborazione della parola.
Tale percorso viene rappresentato per intero nella propria colonna di sinistra ma anche come evidenziazione in blu degli archi che sono stati attraversati.

Infine, abbiamo inserito le funzioni di salvataggio e apertura dei grafi in alto a sinistra nel pulsante "File". Al suo fianco troviamo "Edit" in cui vengono forniti strumenti ulteriori per la modifica del grafo.

Di seguito vari esempi di prompt del sistema che richiede input dall'utente:



Algoritmo Chiave

Un algoritmo su cui vale la pena approfondire è quello usato per l'esecuzione di WordAutomata.

Questo algoritmo, fornito l'automa e la parola, va a generare il l'esito, il percorso e l'eventuale parola restante in caso di non accettazione della parola.

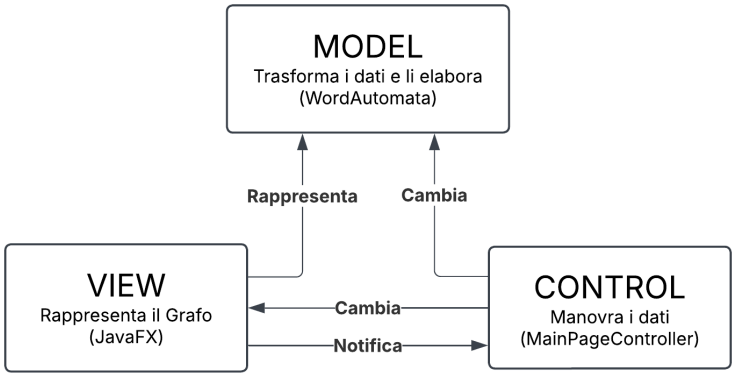
Nello scrivere l'algoritmo ci si è basati sulla teoria dei grafi e si è utilizzato la nozione della matrice di adiacenza. Tale matrice rappresenta l'intero grafo e torna molto utile all'interno di un loop.

```
while (!remainingWord.isEmpty()) {
    found = false;
    for (int i = remainingWord.length(); i > 0; i--) {
        subWord = remainingWord.substring(beginIndex:0, i);
        for (tempInd = 0; tempInd < stateNum && !found; tempInd++)
            found = graphMatrix[currStateInd][tempInd].contains(subWord);
        if (found) {
            stateHistory.add(stateList.get(currStateInd).name);
            stringHistory.add(subWord);
            remainingWord = remainingWord.replaceFirst(subWord, replacement:"");
            currStateInd = tempInd - 1;
            break;
        }
    }
    this.remainingWord = remainingWord;
    if (!found) return false;
}
stateHistory.add(stateList.get(currStateInd).name);

return stateList.get(currStateInd).isFinal;
```

Dopo un'analisi veloce si può dedurre che l'algoritmo ha un costo computazionale massimo di $O(l^2 n)$ dove l sta per il numero di caratteri nella parola e n sta per il numero di nodi. Tale costo può essere moltiplicato dal costo di altre funzioni usate come contains() o substring().

Pattern Architetturale



Il pattern architetturale scelto è stato quello dell'MVC, ovvero Model View Control.

Questo pattern permette la separazione di tre attività del sistema:
Model: si tratta della parte di elaborazione e interpretazione dei dati. View: gestisce la rappresentazione grafica dei dati. Control: è la parte che gestisce l'interazione con l'utente e in base a queste va a cambiare le informazioni delle due componenti spiegate sopra.

I componenti del gruppo si sentivano più abili con il linguaggio Java piuttosto che con altri e dato che la libreria JavaFX è stata introdotta a lezione, si è ritenuto ragionevole scegliere questo pattern.

Di conseguenza solo due aspetti del sistema su tre sono stati da sviluppare, risparmiando quindi costi di sviluppo grazie a tale scelta. Abbiamo quindi deciso di suddividerci i due compiti rimanenti.

Il mantenere separate questi tre aspetti del software implica vantaggi notevoli per quanto riguarda la semplicità del codice in quanto ogni sezione si specializza nel compito affidatole. In particolare, abbiamo riscontrato un aumento della facilità di scrittura del codice in quanto i ruoli erano ben definiti.

Design Pattern

I Design Pattern sono stati implementati in una maniera per niente standard in quanto non sono state create classi apposite. Quindi, citiamo di seguito pezzi di codice che richiamano gli stessi concetti.

Adapter

Si tratta di un design pattern di tipo strutturale. Questo è come lo abbiamo implementato:

Per effettuare l'esecuzione dell'automa vanno adattate le liste di Node e di Edge in una sola lista di State. Questo perché l'algoritmo scritto usa una matrice, la quale è utilizzabile solo quando un nodo è a conoscenza degli archi che partono da sé stesso.
Siamo consci che il design pattern in questione prevede la creazione di un oggetto separato ma nel nostro caso non era indispensabile; pertanto, abbiamo deciso di aggiungerlo comunque.
Vi è quindi un metodo in WordAutomata chiamato "translate()" che va a fare tale adattamento:

```
private ArrayList<State> translate() {
    ArrayList<State> temp = new ArrayList<>();

    for(Node n : listNode) {
        State s = new State(n.getName(), n.isNodeInitial(), n.isNodeFinal());
        for(Edge e : listEdge)
            if(e.getNodes()[0].getName().equals(s.name))
                s.edges.put(e.getValue(), e.getNodes()[1].getName());
        temp.add(s);
    }

    return temp;
}
```

Inoltre, per evitare loop di puntatori la lista di transizioni appartenente a State è stata resa una HashMap di coppia di stringhe in cui si memorizzano i valori delle transizioni e il nome dei nodi a cui puntano. Questo funziona perché ogni nodo è univocamente identificato dal proprio nome e perché non possono esserci transizioni con lo stesso valore che partono dallo stesso nodo.

Iterator

Si tratta di un design pattern Comportamentale. Questo è come lo abbiamo implementato:

Questo metodo viene usato durante l'apertura di un grafo da un file e usa l'iterator della classe ArrayList<> per aggiungere ogni Node e ogni Edge alla rappresentazione grafica del software.

Per implementarlo propriamente avremmo dovuto implementare la classe Iterator alle nostre classi Node ed Edge e sovrascrivere propriamente i necessari metodi.

```
private void updateGraphPane() {
    for (Edge edge: edgeList) {
        Edge.edgeList = edgeList;

        graphPane.getChildren().add(edge.getGroup());
        edgeMenuList.getChildren().add(edge.getStackPane());

        setContextMenuEdge(edge);
    }

    for (Node node: nodeList) {
        Node.nodeList = nodeList;
        node.setController(this);

        graphPane.getChildren().add(node.getGroup());
        nodeMenuList.getChildren().add(node.getStackPane());

        setContextMenuNode(node);
    }
}
```

Memento

Un design pattern che avremmo voluto implementare è Memento. Questo design pattern memorizza gli stati passati di un'istanza di una classe. Sarebbe stato perfetto per implementare un sistema di disfacimento e ripetizione delle azioni prese dall'utente nella modifica del grafo.

Purtroppo, per mancanza di tempo non siamo riusciti ad implementare la funzionalità ma abbiamo pensato di citare comunque il design pattern in quanto abbiamo già pensato a come implementarlo.

Strategia di Sviluppo

All'inizio abbiamo pensato che importare una libreria prefabbricata per la rappresentazione dei grafi fosse una buona idea in quanto avremmo sfruttato codice già esistente e ci saremmo potuti concentrare sullo sviluppo di altre funzionalità.

Abbiamo però cambiato idea in quanto la libreria in questione era fin troppo complessa per il nostro utilizzo. Per questo motivo abbiamo optato per lo sviluppo ex novo della rappresentazione grafica.

Extreme Programming

La strategia di sviluppo utilizzata è stata prevalentemente di tipo Extreme Programming.

Il codice è stato controllato ad ogni passo, assicurandosi che rimanesse pulito e leggibile. Ogni cambiamento ha scaturito il controllo che tutte le funzionalità fossero ancora correttamente funzionanti. Sono stati fatti miglioramenti di codice o implementazioni di nuove funzionalità ogni giorno.
Le differenze tra la nostra strategia e quella dell'Extreme Programming è che non ci sono stati cambiamenti di requisiti; i requisiti sono stati implementati gradualmente. Inoltre, il cliente non era presente e anzi abbiamo preso noi stessi il suo ruolo cercando di seguire i requisiti alla lettera.
Un'altra differenza sta nel fatto che prima di implementare una funzionalità importante abbiamo sempre trovato insieme il metodo di implementazione migliore e quindi stilato un piano su cui basarci durante lo sviluppo.

Pair Programming

Vista la dimensione del gruppo era un peccato non utilizzare la strategia del Pair Programming.

Questa strategia prevede lo sviluppo parallelo da parte di due sviluppatori, i quali si supportano a vicenda e discutono su come implementare certe funzionalità. In particolare, ci sono due ruoli: quello del conducente, che scrive codice, e quello dell'osservatore, che supervisiona il conducente e gli propone correzioni.

Ci siamo spesso anche scambiati i ruoli in base a quale parte del sistema stava venendo sviluppata.

Strategie di Testing

Per ogni classe o componente singola del sistema è stato scritto del codice che andava a testarne il corretto funzionamento. Nel momento in cui invece il software è stato completato e tutte le funzionalità erano presenti, abbiamo ritenuto che il testing manuale fosse preferibile. Questo perché il funzionamento del software si basa pesantemente sull'interazione grafica con l'utente.

Per questo abbiamo creato vari file di grafi su cui abbiamo testato il comportamento del software sia per la parte di costruzione del grafo che per la parte di esecuzione dell'automa.

In ogni caso, il testing non è stato limitato solo a noi sviluppatori ma anche a persone esterne. In particolare, abbiamo scelto sia persone dotate sia persone prive di una certa formazione informatica.

Test Sviluppatori

Questi tipi di test sono stati effettuati prima del completamento dell'interfaccia grafica del software. Si tratta del controllo del comportamento dei singoli metodi di ogni classe. Questo è avvenuto dopo ogni sviluppo di tali metodi.

Grazie a questi test, a completamento dell'interfaccia grafica, gli errori osservati sono stati di gran lunga inferiori a quelli che si sarebbero osservati.

Test Utenti con Familiarità agli Automi

Questi test sono avvenuti attraverso l'utilizzo del software in ogni sua parte per creare ed eseguire automi anche complessi. Grazie a loro sono state rivelate funzionalità mancanti e anche errori grafici o di logica.

Per esempio, un utente ha provato a creare un grafo che necessitava di avere nodo inziale e finale corrispondenti. Tale test ha evidenziato il problema e ci ha condotti a prendere la scelta commentata in dettaglio nelle scelte progettuali.

Test Utenti Generici

Questi test hanno avuto lo scopo di misurare l'intuitività del software proposto in quanto gli errori erano già stati trovati dagli utenti precedenti.