Università degli Studi di Verona

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Ingegneria del Software

Documentazione di progetto — Word Automata

Studenti

Lorenzo Di Berardino	VR487434
Mateo Gjika	VR488633
Filippo Milani	VR486588

Docenti

Carlo Combi
Pietro Sala

A.A. 2023 – 2024

Indice

1 Requisiti

1.1 Casi d'uso

L'utilizzo del programma realizzato non prevede alcuna distinzione di ruoli del fruitore. Pertanto, in tutti i diagrammi e specifiche dei casi d'uso che seguono, l'unico attore coinvolto è l'utente generico, ivi indicato come *User*. I termini specifici relativi alle componenti dell'automa sono spiegati in *Nomenclatura*, sezione relativa allo *Sviluppo*, fase in cui sono stati introdotti per la prima volta.

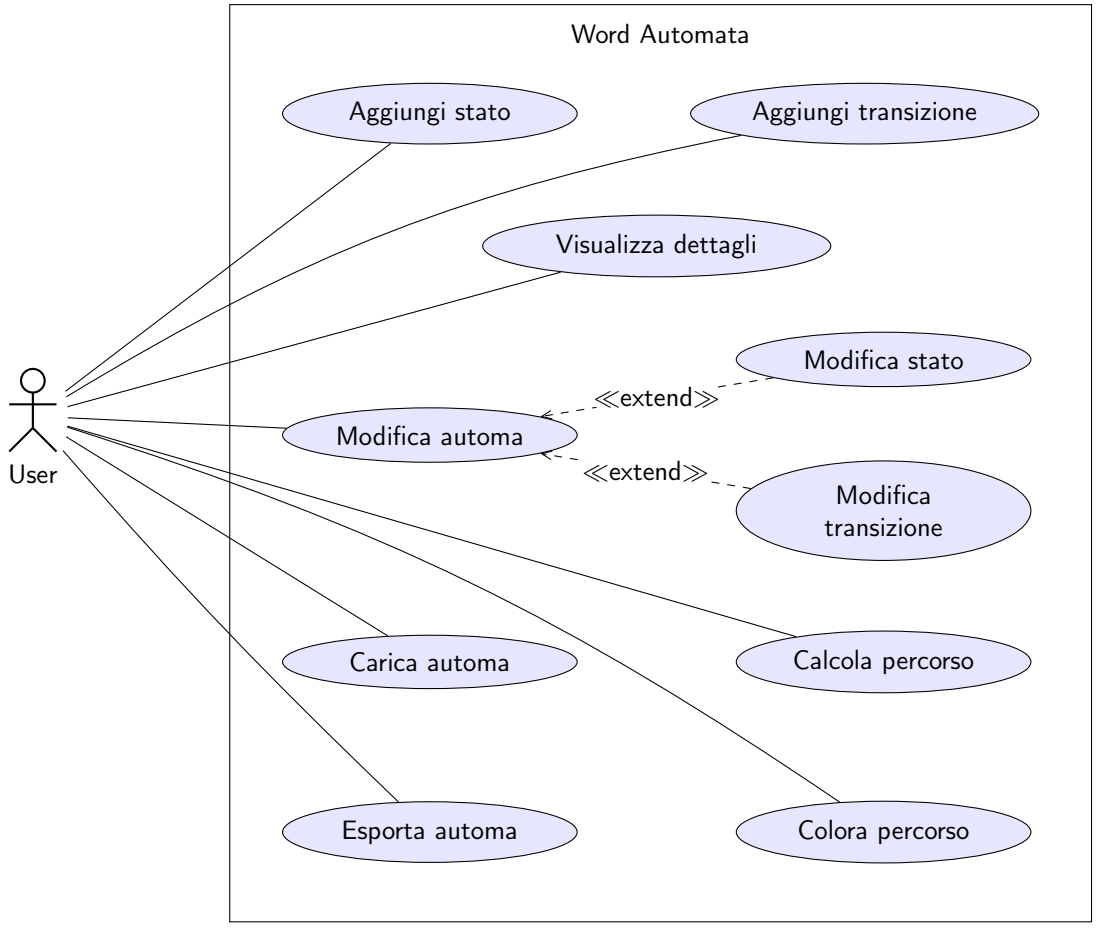


Figura 1.1: Diagramma generale dei casi d'uso.

1.2 Specifiche dei casi d'uso

Vengono ora espansi i singoli casi d'uso individuati, illustrando, per quelli ritenuti di maggiore rilevanza, il diagramma di interazione associato o quello di attività, a seconda del caso.

Caso d'uso	**Aggiungi stato**
Attore	User
Descrizione	L'utente aggiunge uno stato all'automa.
Precondizioni	Il programma è avviato.
Sequenza degli eventi	1. L'utente seleziona l'opzione per aggiungere uno stato. 2. L'utente inserisce le informazioni sullo stato (etichetta, è finale). 3. Se esiste già uno stato con la stessa etichetta, viene mostrato un messaggio di errore, invitando a riprovare.
Postcondizioni	Lo stato è stato aggiunto all'automa.

Tabella 1.1: Specifica del caso d'uso *Aggiungi stato*.

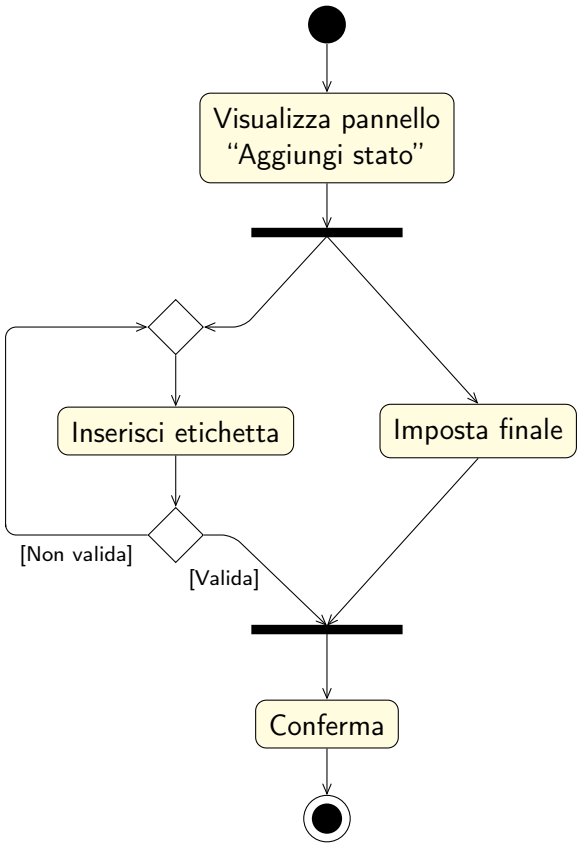


Figura 1.2: Diagramma di attività *Aggiungi stato*.

Caso d'uso	**Aggiungi transizione**
Attore	User
Descrizione	L'utente aggiunge una transizione all'automa.
Precondizioni	È stato inserito almeno uno stato.
Sequenza degli eventi	1. L'utente seleziona l'opzione per aggiungere una transizione. 2. L'utente inserisce le informazioni sulla transizione (etichetta, stato di partenza, stato di arrivo). 3. Se esiste già una transizione con le stesse informazioni, viene mostrato un messaggio di errore, invitando a riprovare.
Postcondizioni	La transizione è stata aggiunta all'automa.

Tabella 1.2: Specifica del caso d'uso *Aggiungi transizione*.

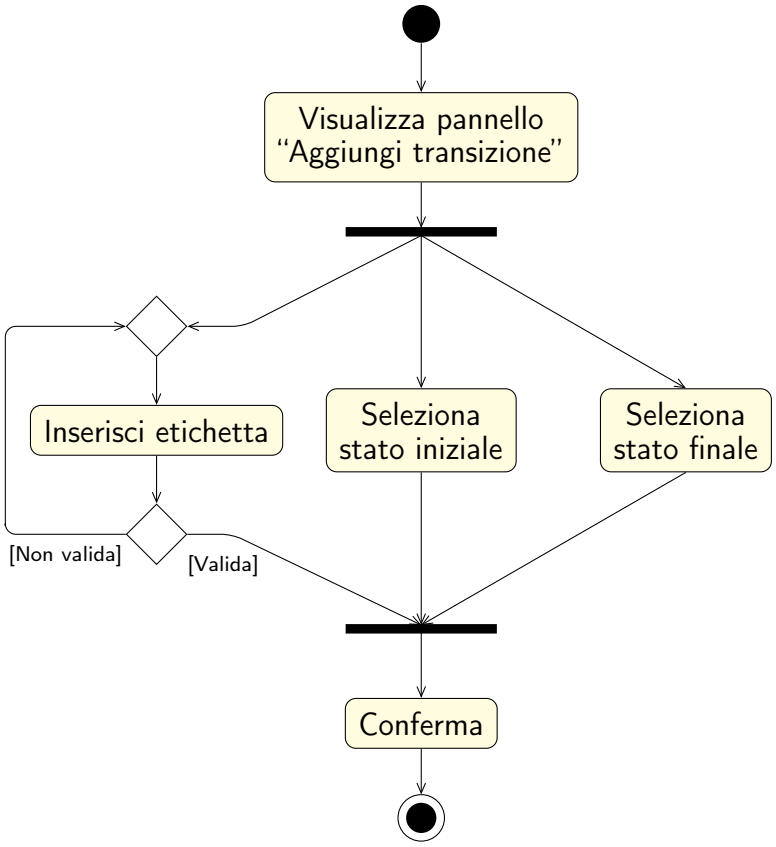


Figura 1.3: Diagramma di attività *Aggiungi transizione*.

Caso d'uso	**Carica automa**
Attore	User
Descrizione	L'utente carica un automa precedentemente salvato.
Precondizioni	Il programma è avviato.
Sequenza degli eventi	1. L'utente seleziona l'opzione per caricare un automa. 2. L'utente sceglie il file da caricare.
Postcondizioni	L'automa è stato caricato.

Tabella 1.3: Specifica del caso d'uso *Carica automa*.

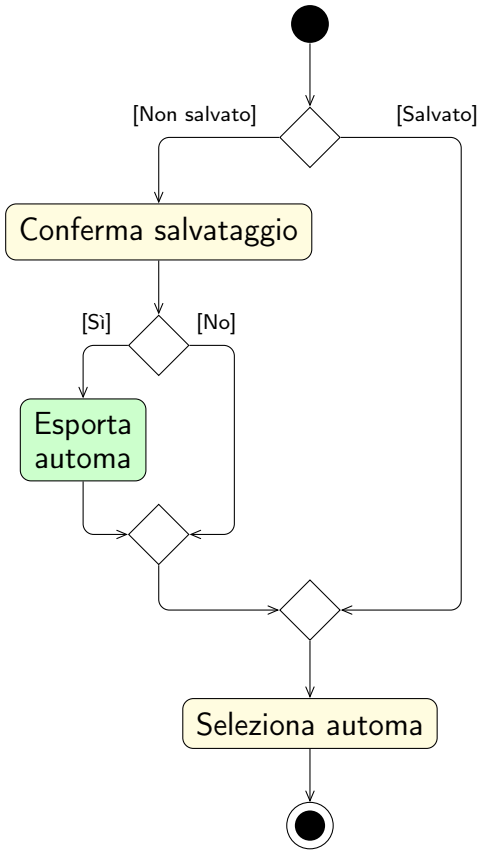


Figura 1.4: Diagramma di attività *Carica automa*.

Caso d'uso	**Esporta automa**
Attore	User
Descrizione	L'utente esporta l'automa in un file.
Precondizioni	È stato disegnato l'automa.
Sequenza degli eventi	1. L'utente seleziona l'opzione per esportare l'automa. 2. Se non è stato già inserito, l'utente sceglie il percorso di salvataggio.
Postcondizioni	L'automa è stato esportato.

Tabella 1.4: Specifica del caso d'uso *Esporta automa*.

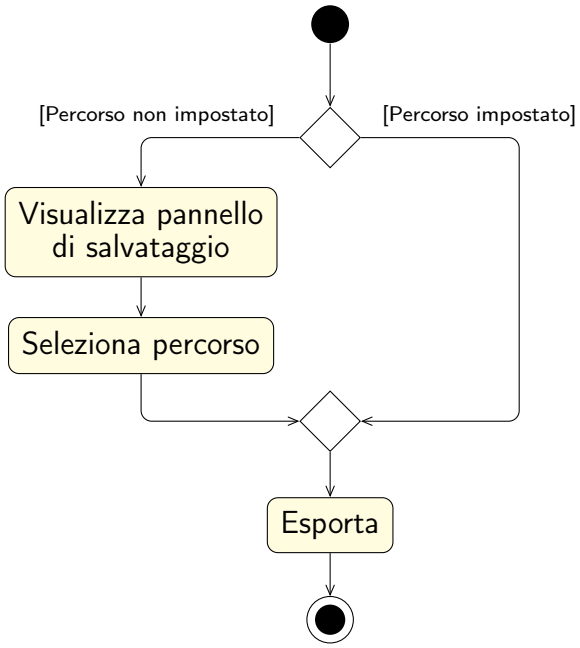


Figura 1.5: Diagramma di attività *Esporta automa*.

Caso d'uso	**Modifica stato**
Attore	User
Descrizione	L'utente modifica uno stato dell'automa.
Precondizioni	È stato inserito almeno uno stato.
Sequenza degli eventi	1. L'utente seleziona uno stato. 2. L'utente modifica le informazioni sullo stato (etichetta, è finale, è iniziale). 3. Se esiste già uno stato con la stessa etichetta, viene mostrato un messaggio di errore, invitando a riprovare.
Postcondizioni	Lo stato è stato modificato.

Tabella 1.5: Specifica del caso d'uso *Modifica stato*.

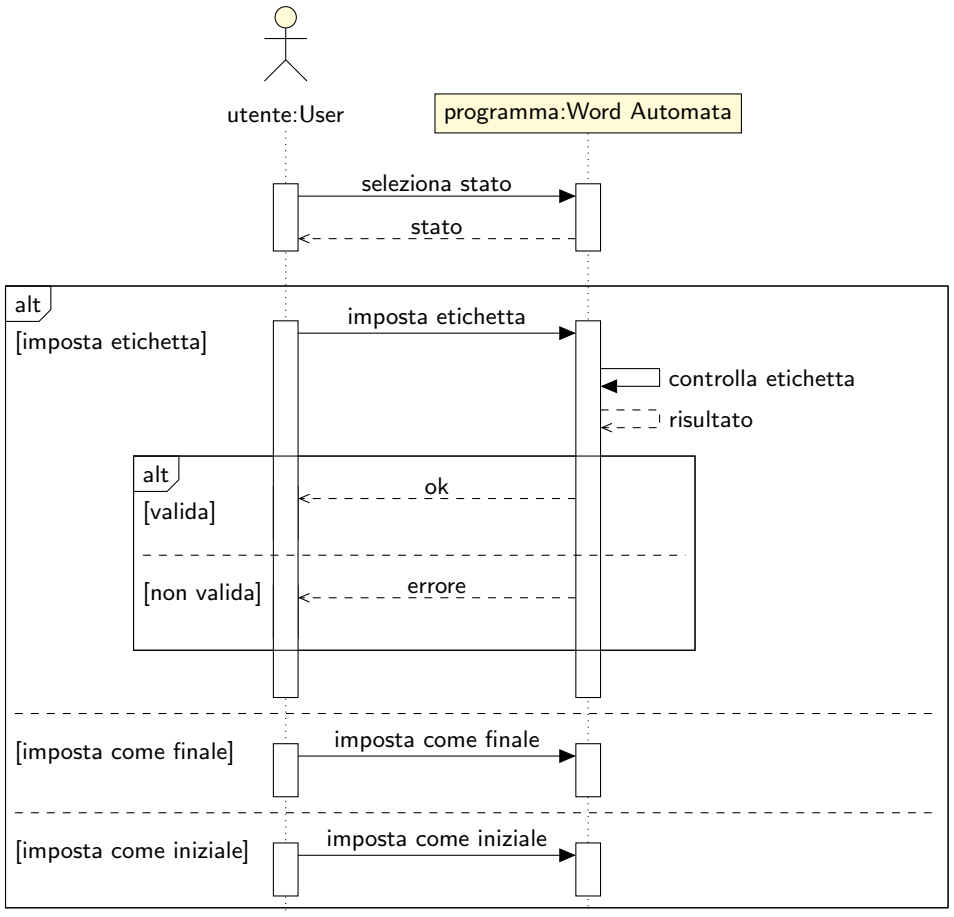


Figura 1.6: Diagramma di sequenza *Modifica stato*.

Caso d'uso	**Modifica transizione**
Attore	User
Descrizione	L'utente modifica una transizione dell'automa.
Precondizioni	È stata inserita almeno una transizione.
Sequenza degli eventi	1. L'utente seleziona l'opzione per modificare una transizione. 2. L'utente sceglie la transizione da modificare. 3. L'utente modifica l'etichetta della transizione.
Postcondizioni	La transizione è stata modificata.

Tabella 1.6: Specifica del caso d'uso *Modifica transizione*.

Caso d'uso	**Visualizza dettagli**
Attore	User
Descrizione	L'utente visualizza le informazioni su stati o transizioni.
Precondizioni	Almeno uno stato o una transizione sono stati inseriti.
Sequenza degli eventi	1. L'utente seleziona l'opzione per visualizzare i dettagli di stati o transizioni. 2. L'utente seleziona l'elemento di cui visualizzare i dettagli.
Postcondizioni	Sono mostrate le informazioni sull'elemento scelto.

Tabella 1.7: Specifica del caso d'uso *Visualizza dettagli*.

Caso d'uso	**Calcola percorso**
Attore	User
Descrizione	Viene calcolato un percorso per una stringa sottoposta all'automa.
Precondizioni	L'automa è stato disegnato.
Sequenza degli eventi	1. L'utente inserisce la stringa per cui calcolare il percorso. 2. Se non esiste un percorso per la stringa fornita, viene mostrato un messaggio di errore.
Postcondizioni	Se esiste, viene mostrato il percorso calcolato.

Tabella 1.8: Specifica del caso d'uso *Calcola percorso*.

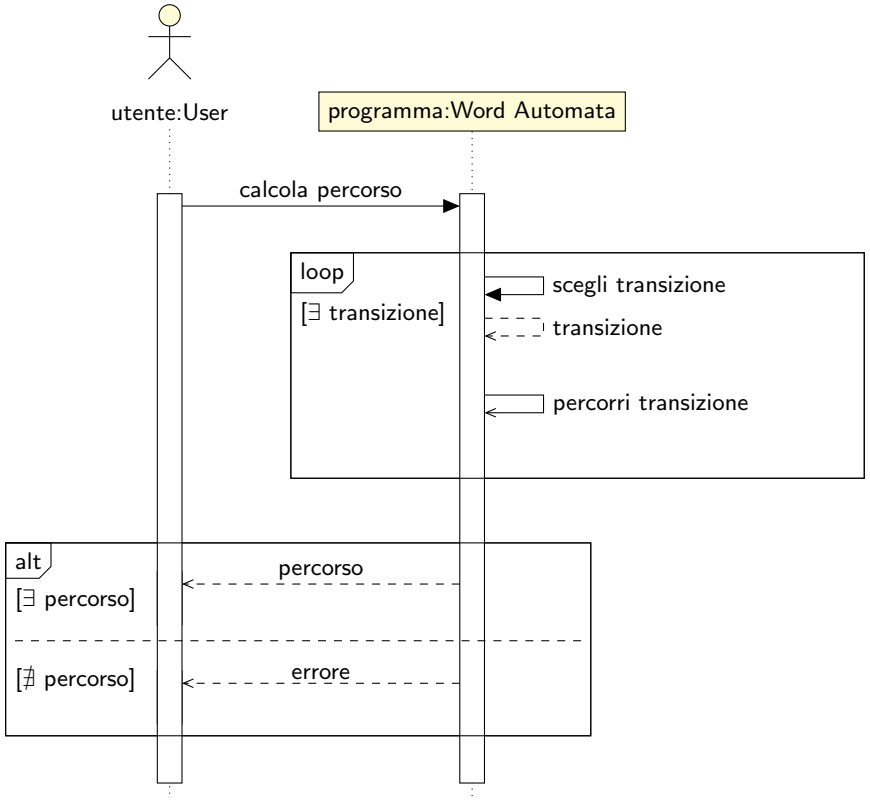


Figura 1.7: Diagramma di sequenza *Calcola percorso*.

Caso d'uso	**Colora percorso**
Attore	User
Descrizione	Viene colorato il percorso calcolato nell'automa.
Precondizioni	Il percorso è stato calcolato.
Sequenza degli eventi	1. L'utente seleziona l'opzione per colorare il percorso.
Postcondizioni	Il percorso è stato colorato.

Tabella 1.9: Specifica del caso d'uso *Colora percorso*.

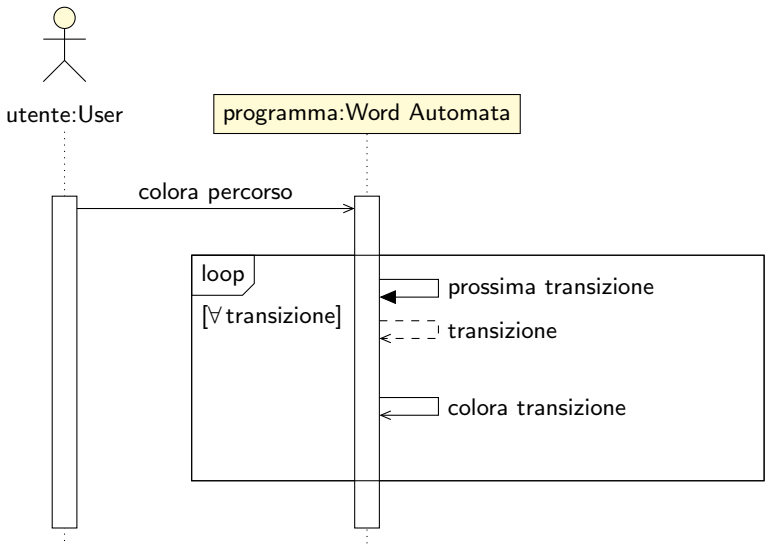


Figura 1.8: Diagramma di sequenza *Colora percorso*.

2 Sviluppo

2.1 Piano di sviluppo e dettagli implementativi

Lo sviluppo del progetto ha visto la combinazione delle logiche *plan-driven* e *agile*, in particolare nella loro declinazione *incrementale*. La scelta di queste metodologie è stata dettata, oltre da passate esperienze del team di sviluppo, anche dalla necessità di garantire un'adeguata pianificazione e controllo del progetto, unitamente alla possibilità di adattarsi a cambiamenti e nuove esigenze che sarebbero potute emergere durante lo sviluppo, come in effetti è avvenuto. Inoltre, l'attività vera e propria di scrittura del codice è avvenuta secondo la modalità di *pair programming*, in modo da garantire una maggiore qualità del codice prodotto e una maggiore condivisione delle conoscenze tra i membri del team di sviluppo.

La prima fase del processo di progettazione ha visto la definizione di un'insieme di prerogative minime, elicitate a partire dalle specifiche presentate, che l'applicazione avrebbe dovuto soddisfare. Queste sono state definite in termini di requisiti funzionali e non funzionali, e sono state utilizzate come base per la redazione di un piano di sviluppo. Questo piano ha previsto l'individuazione di un'architettura software, la scelta delle tecnologie da utilizzare e la definizione di un insieme di milestone, che avrebbero scandito lo sviluppo del progetto. Le prerogative minime raccolte sono state in seguito tradotte in un insieme di specifiche tecniche, che hanno guidato il team di sviluppo nella realizzazione del codice. La separazione dei compiti è stata eseguita, dapprima, raggruppando le attività per area di appartenenza, quali *front-end* e *back-end*, e in seguito procedendo, in base alla necessità, alla risoluzione di singole unità di lavoro. Questo approccio ha permesso al team di sviluppo di lavorare in parallelo su più fronti, garantendo una maggiore efficienza e una maggiore flessibilità nel caso di cambiamenti interni emersi durante lo sviluppo.

La produzione del codice è stata effettuata mediante l'utilizzo di un sistema di versioning distribuito, `Git`, che ha permesso al team di sviluppo di lavorare in parallelo sulla medesima *codebase*, mantenendo un'adeguata tracciabilità delle modifiche effettuate. La repository del progetto è raggiungibile a questo indirizzo. In particolare, il codice è stato scritto in linguaggio `Java`, utilizzando il framework `JavaFX` per la realizzazione dell'interfaccia grafica. L'effettiva scrittura è avvenuta mediante l'IDE `Visual Studio Code`.

La fase di produzione del programma, inoltre, è stata in larga parte affiancata da una fase di test per i singoli componenti (si veda *Verifica e validazione – Test interni*), in modo da garantire la correttezza del codice prodotto e la conformità alle specifiche. Dopo aver raggiunto un livello di qualità del codice e di operatività del programma soddisfacente per il team di sviluppo e per alcuni *alpha tester* selezionati, è seguita un'operazione sia di ampliamento che, in alcuni casi, di semplificazione delle funzionalità dell'applicazione, unitamente ad un processo di *refactoring* e *decluttering* del codice.

La redazione della documentazione, infine, è stata eseguita in seguito alla conclusione della fase di sviluppo, per mantenere un certo grado di correttezza e completezza delle informazioni riportate. Questa posticipazione è stata giustificata dalla necessità di garantire la coerenza tra il codice prodotto e la documentazione, e di evitare la duplicazione di informazioni. In aggiunta a ciò, alcuni dei

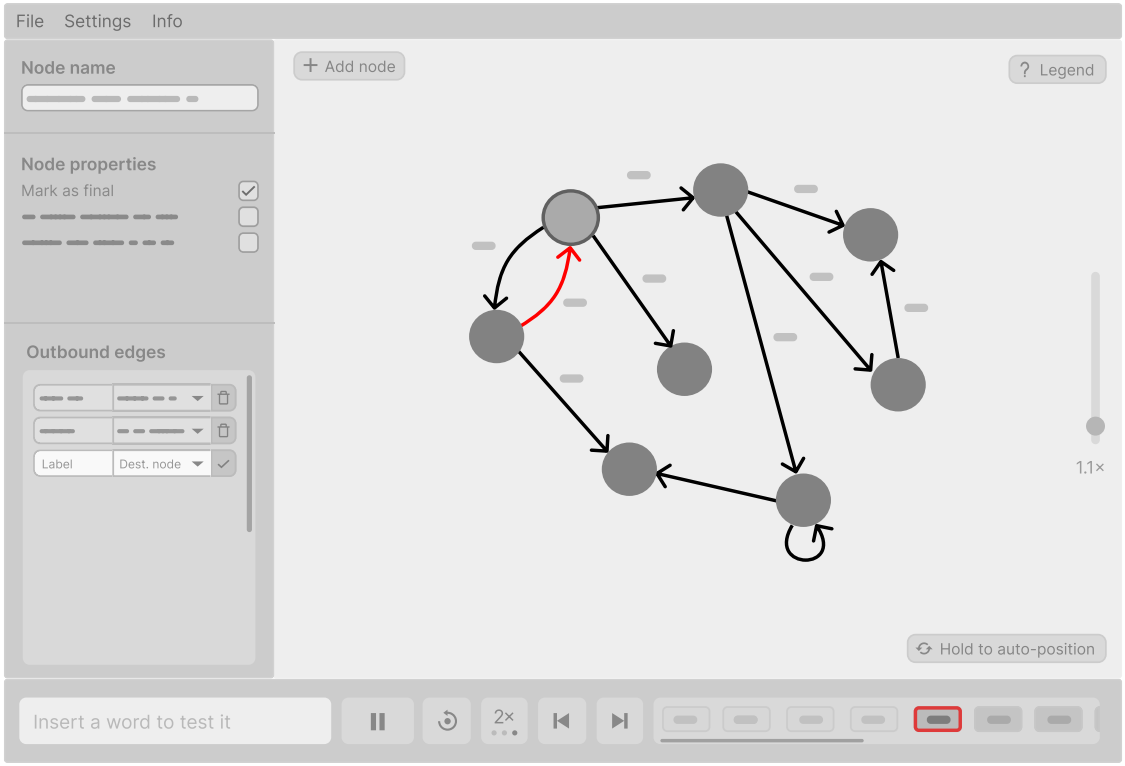


Figura 2.1: Una bozza iniziale dell'interfaccia grafica del programma.

diagrammi qui riportati sono stati realizzati mediante strumenti automatici, i quali necessitavano di un codice sorgente piuttosto completo per poter essere generati correttamente e senza dover ricorrere a modifiche o integrazioni successive.

2.2 Pattern architetturali e di design

La natura del progetto e del framework utilizzato per l'interfaccia grafica, `JavaFX`, hanno influenzato la scelta dei pattern architetturali e di design da utilizzare. Di seguito quelli utilizzati:

Model-View-Controller L'architettura principale del programma, soprattutto relativamente al *front-end*, ha visto l'adozione del pattern *Model-View-Controller* (MVC). Questa scelta è giustificata dalla presenza di una chiara separazione tra dati, logica di presentazione e logica di controllo: l'automa è costituito da un grafo di stati e transizioni, che costituiscono il modello dell'applicazione; la vista è costituita dall'interfaccia grafica, che permette all'utente di interagire con l'automa, ad esempio aggiungendo o rimuovendo stati o transizioni; il controller, infine, si occupa di gestire gli eventi generati dall'interfaccia grafica e di aggiornare il modello di conseguenza.

È stato scelto di non far comunicare direttamente il modello e la vista, ma di far avvenire questo passaggio di informazioni attraverso il controller, poiché l'utilizzo di `JavaFX` richiede che la modifica degli elementi grafici avvenga all'interno del *JavaFX Application Thread*, e il controller è l'unico componente che ha accesso a tale thread.

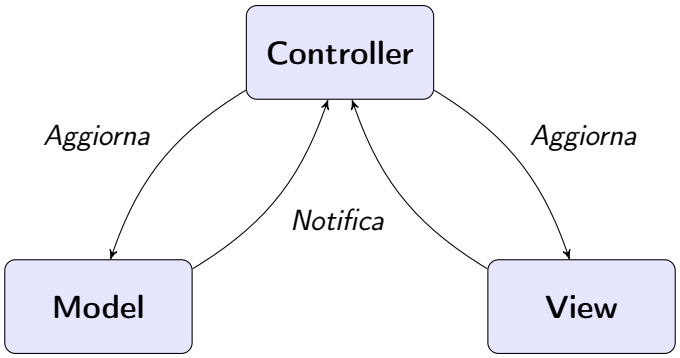


Figura 2.2: Variante del pattern architetturale *Model-View-Controller* adottata.

Registry Le classi che costituiscono il *Controller* necessitano di accedere a una classe che le raggruppa. Questo consente l'invocazione di metodi da parte di altre classi e facilita la modifica dei singoli componenti grafici da parte di altre entità. Per realizzare ciò, invece di posizionare in ognuna delle classi coinvolte un riferimento ai diversi componenti grafici, è stato utilizzato il pattern *Registry* in una sua variante semplificata, dato che nell'ambito del programma non è necessario modificare o eliminare le istanze dei singoli componenti, ma solo crearli e accedervi.

Singleton Un altro design pattern utilizzato è il *Singleton*, per garantire che esista una sola istanza dello stato dell'automa (incluso nella classe `Model`).

Iterator La ricerca di un percorso nell'automa produce un `ListIterator` di transizioni (si veda *Nomenclatura*). Questo iteratore è, come suggerisce il nome, l'implementazione del pattern *Iterator*, che permette di scorrere la lista di transizioni in modo sequenziale e di accedere ai singoli elementi.

2.3 Nomenclatura

Nell'ambito del progetto, si fa riferimento a *Stato* come ai singoli punti di approdo dell'automa. Uno stato può essere *Iniziale*, se è lo stato di partenza dell'automa, *Finale*, se è uno stato di accettazione dell'automa, o entrambi, se è sia iniziale che finale. Una *Transizione* è un collegamenti tra due stati (o lo stesso stato se si tratta di un autoanello). Un *Percorso*, invece, è una sequenza di transizioni. Un *Automa* è dunque costituito da un insieme di stati e transizioni, e da percorsi che collegano stati.

Si dice che esiste un percorso tra due stati se esiste una sequenza di transizioni che collega i due stati. Una parola di input è accettata dall'automa se esiste un percorso che parte dallo stato iniziale e termina in uno stato finale, e tale percorso è costituito da transizioni che consumano porzioni della parola di input, senza lasciare alcuna porzione non consumata al termine del percorso.

2.4 Diagrammi delle classi

Nelle pagine che seguono vengono riportate le principali classi con i loro campi e metodi più rilevanti. Una lista completa, dettagliata e commentata, è disponibile per la visione nel `javadoc` del progetto, oppure accessibile tramite browser a questo indirizzo.

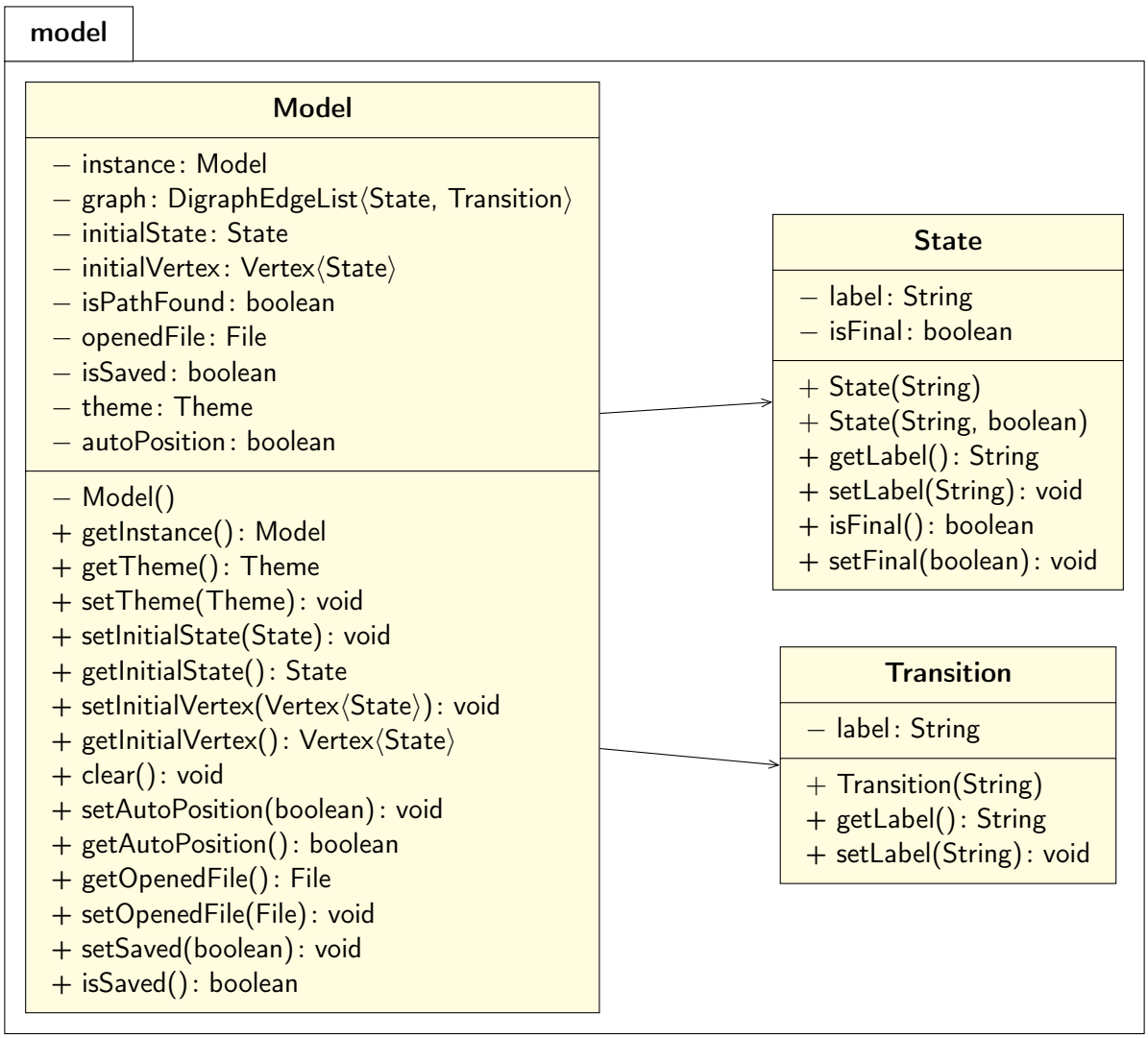


Figura 2.3: Diagramma delle classi relativo al package `model`.

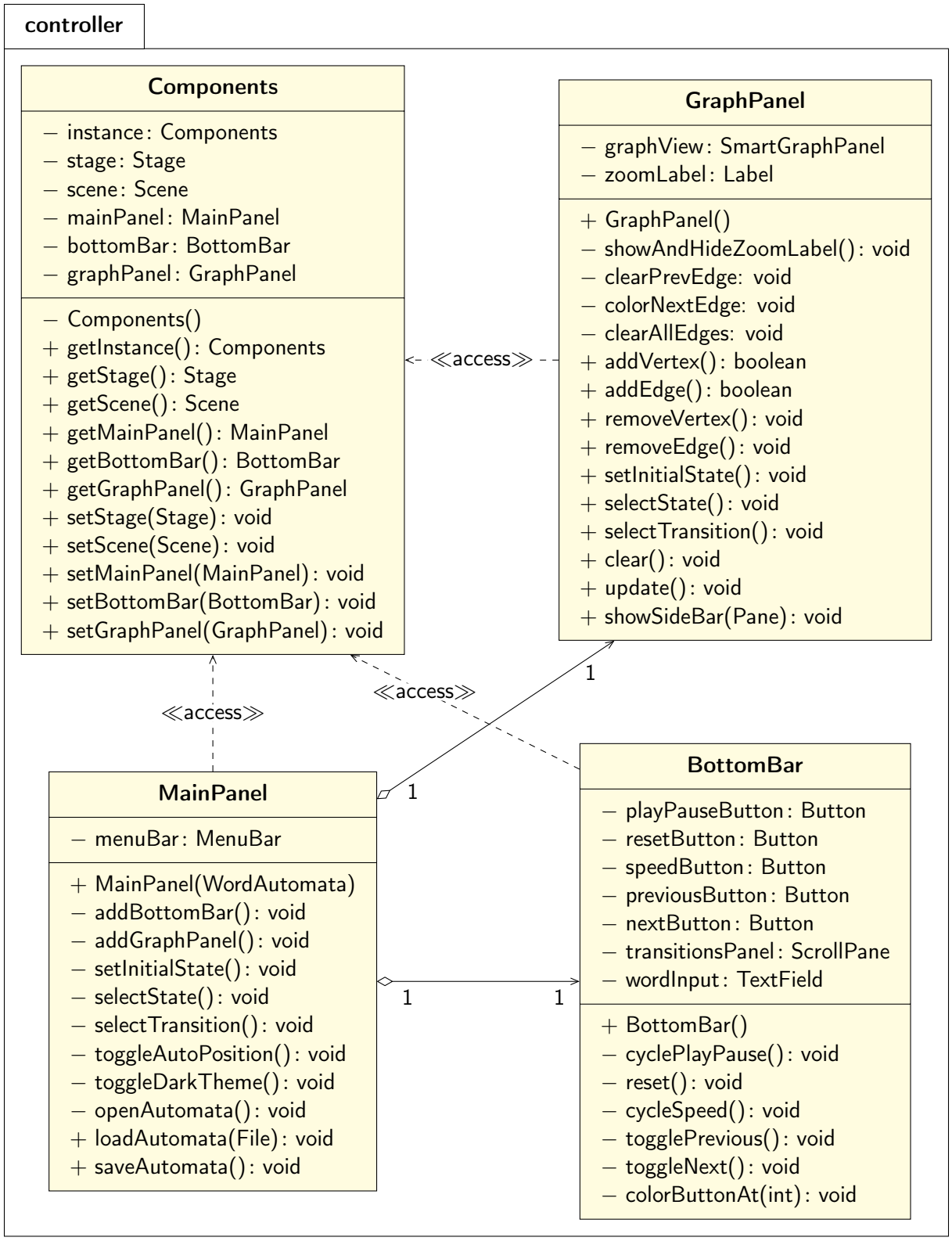


Figura 2.4: Diagramma delle classi relativo al package `controller`.

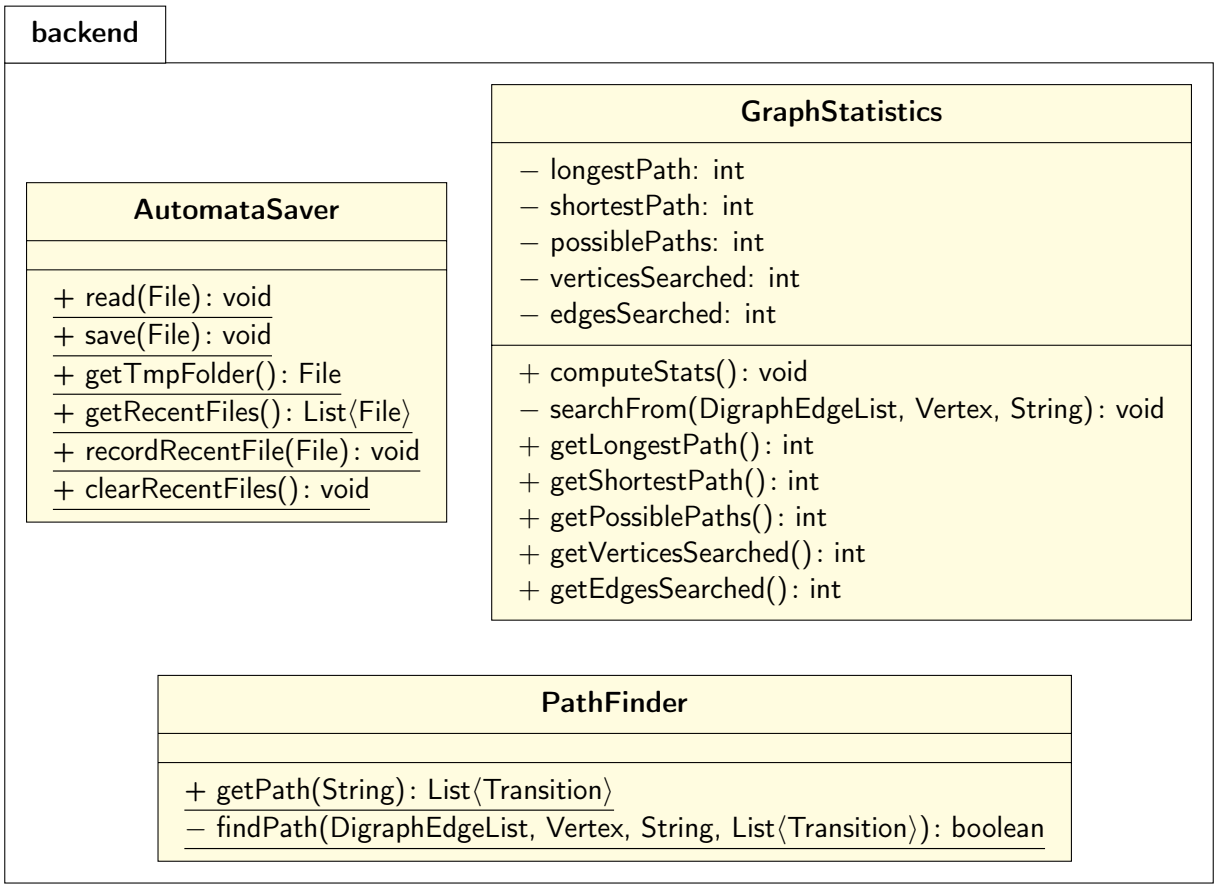


Figura 2.5: Diagramma delle classi relativo al package `backend`.

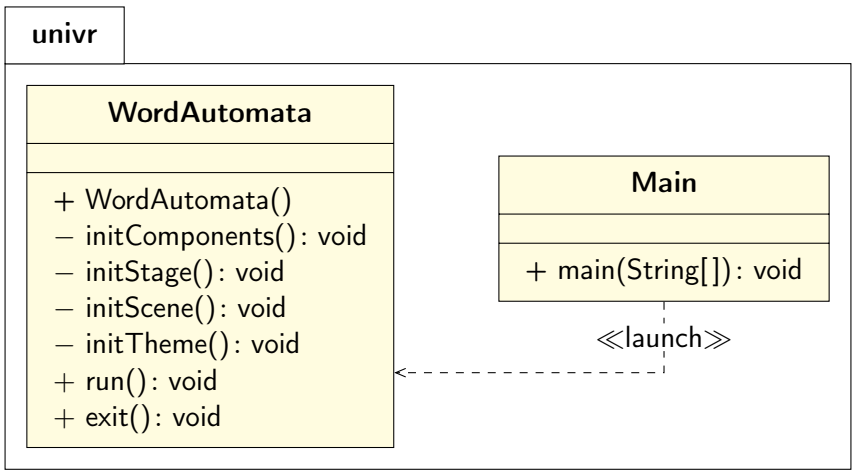


Figura 2.6: Diagramma delle classi relativo al package globale `univr`.

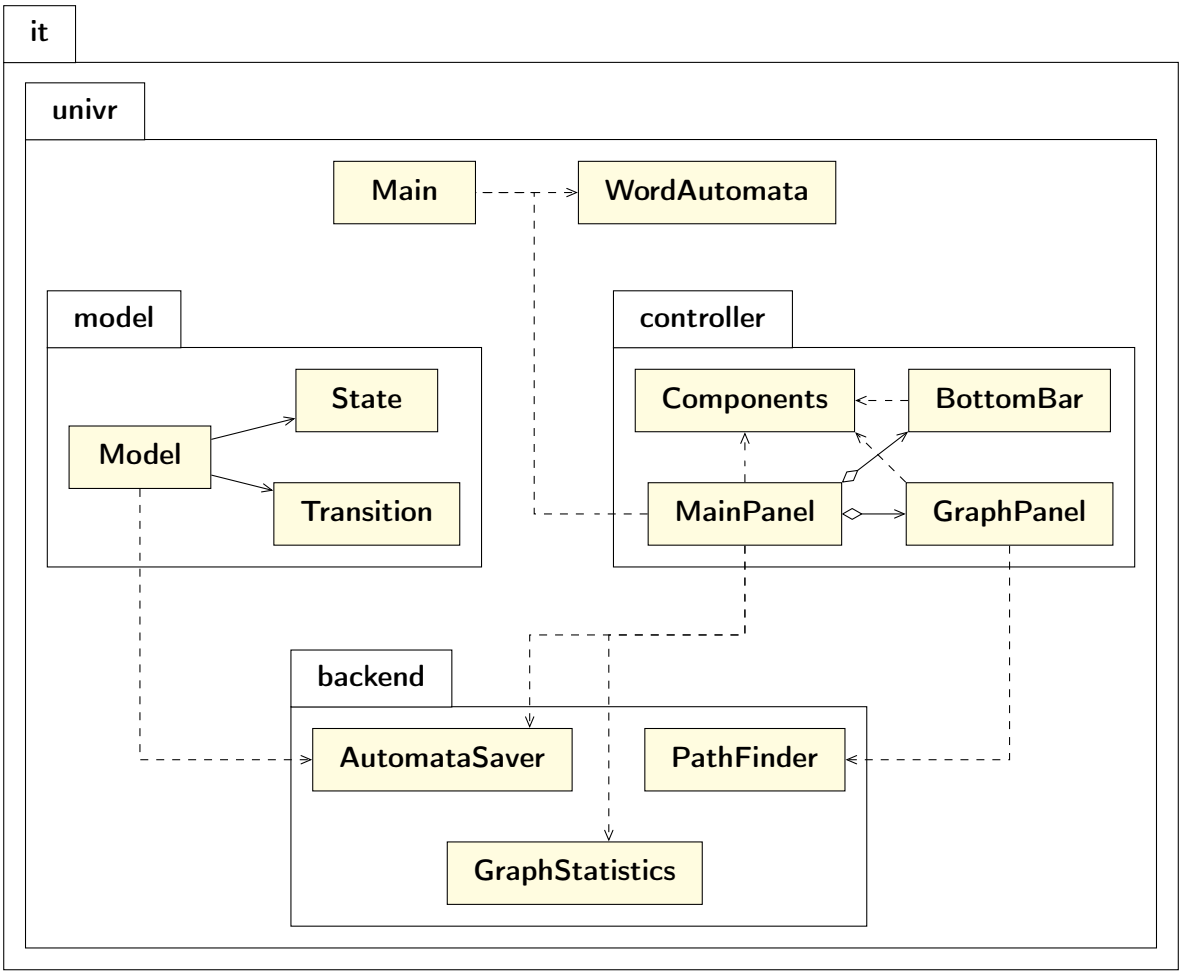


Figura 2.7: Diagramma delle classi condensato.

2.5 Diagrammi di sequenza

Vengono ora presentati i diagrammi di sequenza per le operazioni più rilevanti del programma. Sono state eseguite talvolta delle semplificazioni delle interazioni tra le classi, sia per garantire una maggiore chiarezza e leggibilità del diagramma che per questioni di spazio.

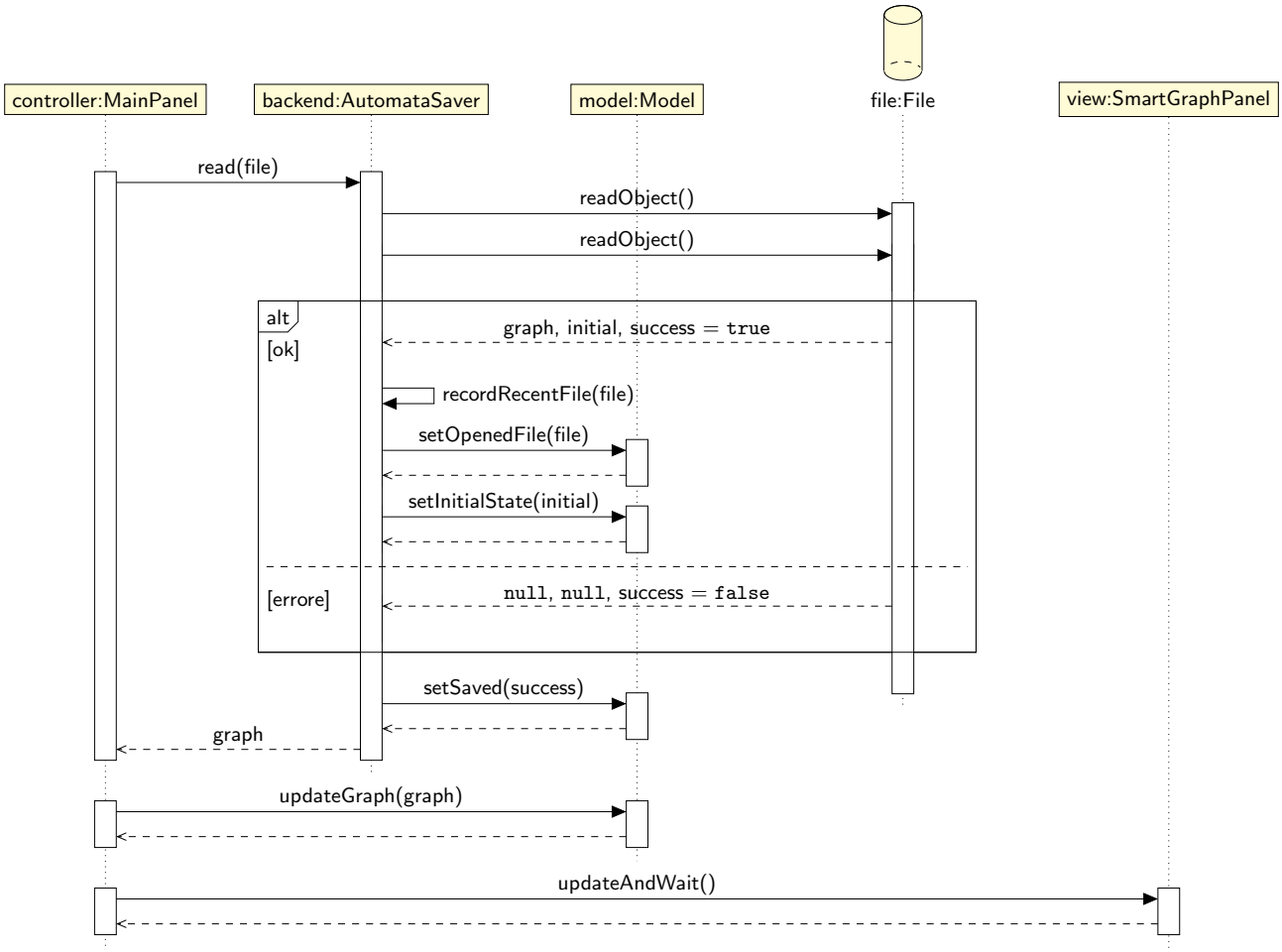


Figura 2.8: Diagramma di sequenza *Importa automa*.

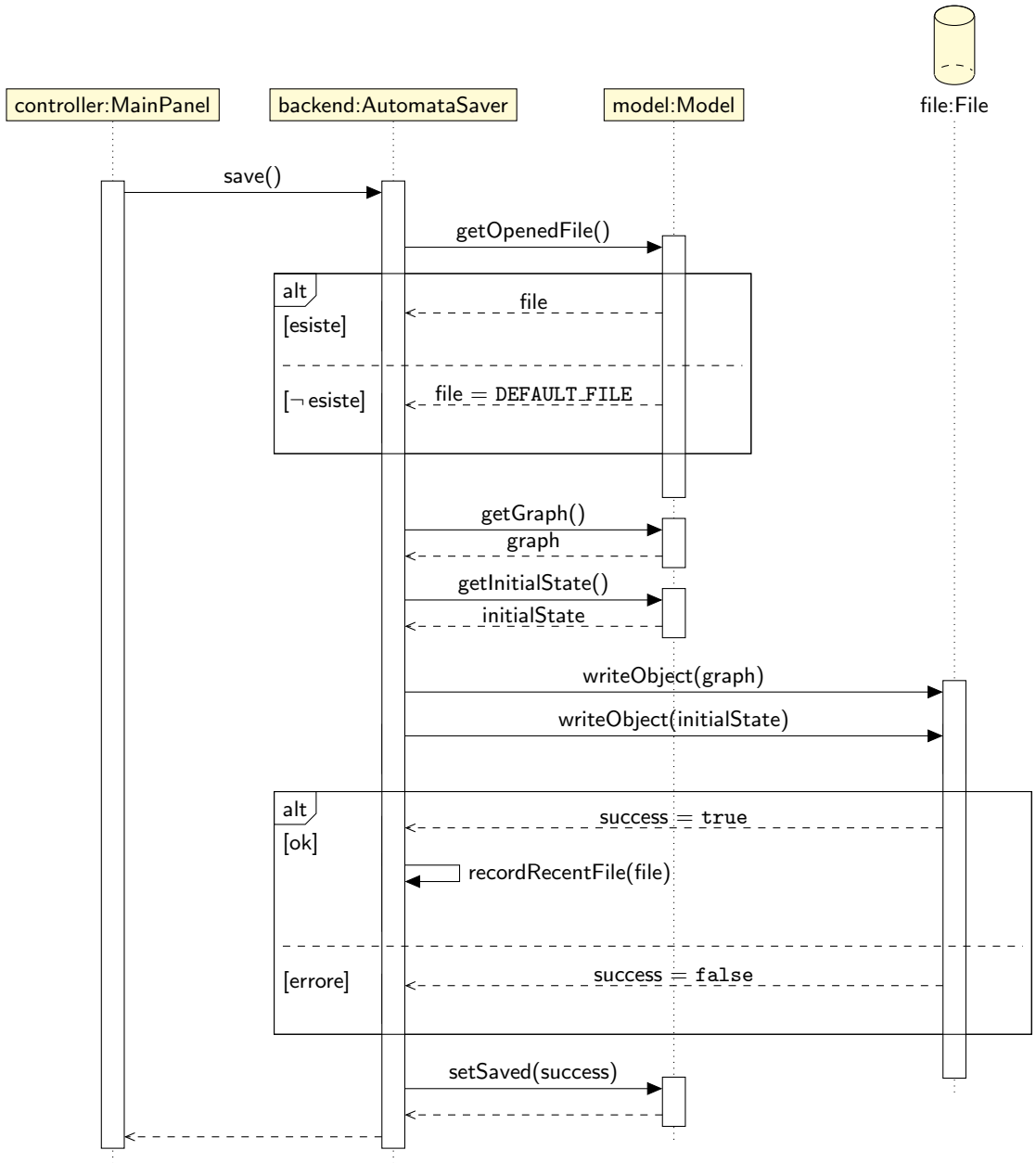


Figura 2.9: Diagramma di sequenza *Esporta automa*.

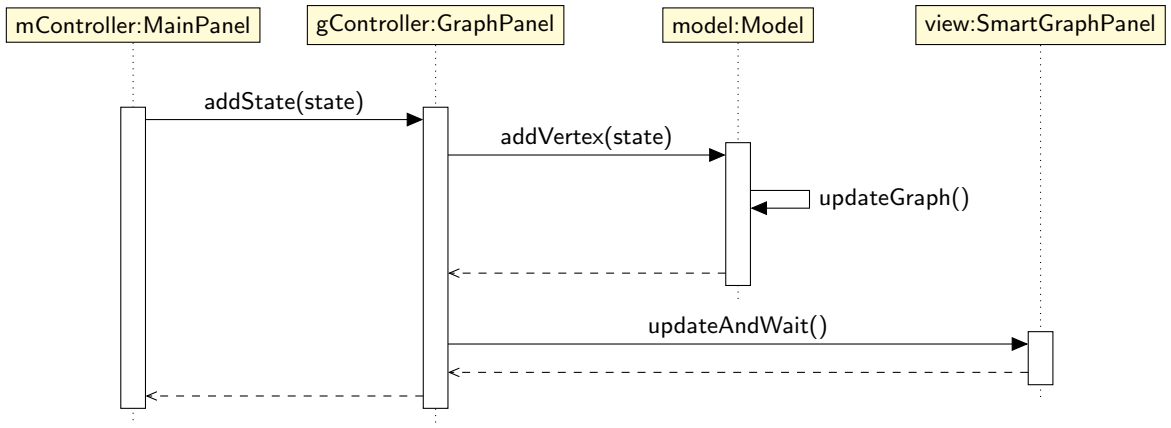


Figura 2.10: Diagramma di sequenza *Aggiungi stato*.

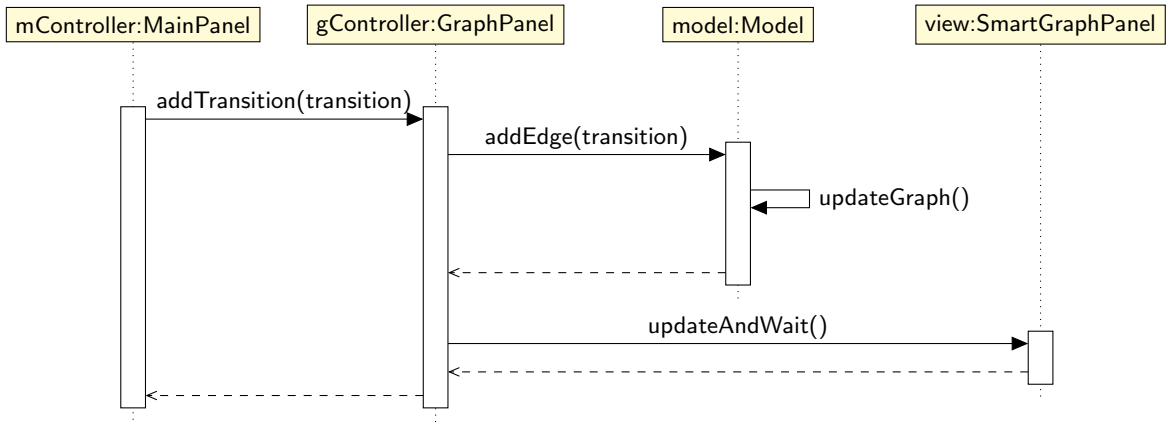


Figura 2.11: Diagramma di sequenza *Aggiungi transizione*.

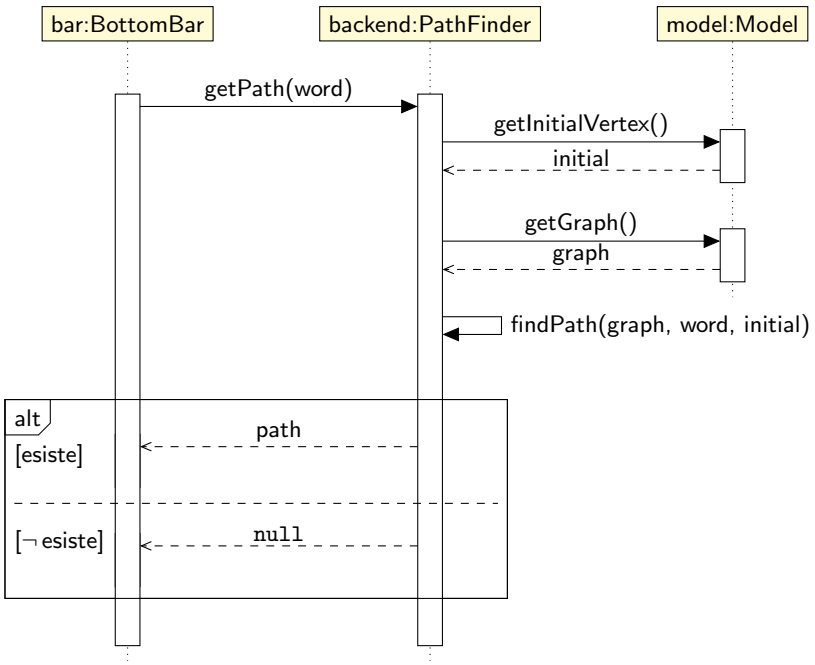


Figura 2.12: Diagramma di sequenza *Calcola percorso*.

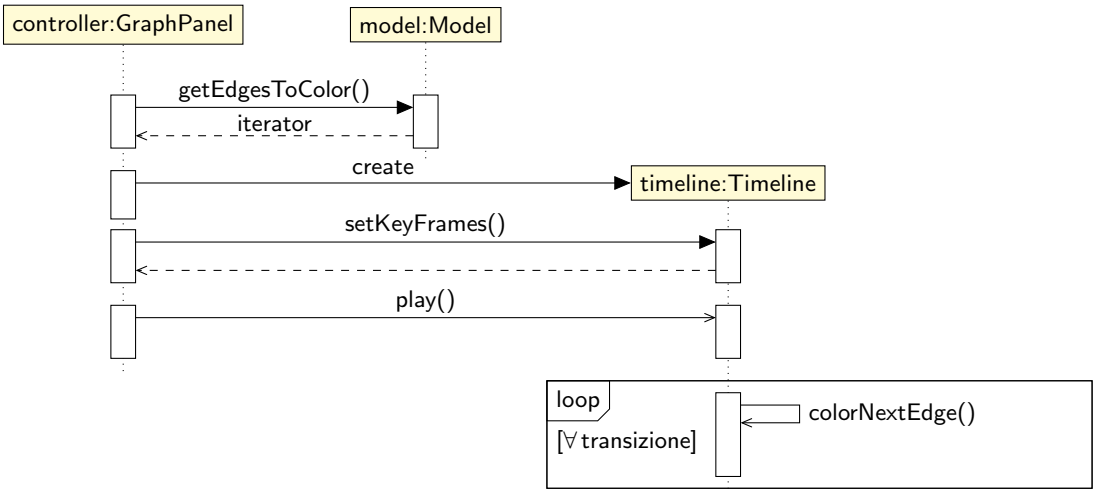


Figura 2.13: Diagramma di sequenza *Colora percorso*.

3 Verifica e validazione

Come anticipato in *Sviluppo*, il processo di verifica e validazione è stato svolto in parallelo con la progettazione e l'implementazione del sistema. In particolare, la verifica è stata effettuata in modo incrementale, ad ogni fase di sviluppo, mentre la validazione è stata effettuata alla fine del processo di sviluppo, una volta completata l'implementazione del programma.

3.1 Test statici

Sono state predisposte ed eseguite due categorie di test: la prima, di tipo statico, ha avuto lo scopo di verificare la correttezza del codice prodotto e la sua conformità alle specifiche; questa parte di test, inoltre, è stata svolta esclusivamente da parte del team di sviluppo ed è stata mirata alla verifica della correttezza del codice prodotto.

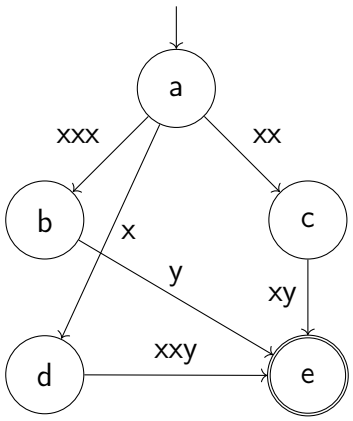
3.1.1 Test di unità

La natura del progetto ha lasciato poco spazio alla possibilità di eseguire test estensivi sul funzionamento delle singole unità, dal momento che è composto da un numero limitato di componenti aventi possibili criticità. Inoltre, per come l'interfaccia grafica è stata costruita, sono state precluse *by design* situazioni di incoerenza o di errore da poter verificare in fase di test. Per questo, è stato possibile eseguire quasi esclusivamente test relativi alla ricerca dei percorsi nell'automa e al salvataggio e caricamento di un automa da file, ovvero operazioni eseguite nel *back-end* del programma.

La realizzazione di questi test è stata effettuata utilizzando il framework `JUnit`. I test sono stati scritti in modo da verificare il corretto funzionamento delle funzionalità implementate, in particolare:

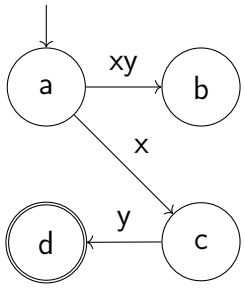
- Per la *ricerca di percorsi* nell'automa, sono stati considerati casi di test relativi a:
  - Accettazione di una stringa da parte dell'automa (percorso esistente)
  - Rifiuto di una stringa da parte dell'automa (parola consumata, ma terminazione su uno stato non finale)
  - Rifiuto di una stringa da parte dell'automa (parola non consumata completamente)
  - In caso di una ricerca con successo, inserimento del percorso trovato nel pannello degli stati all'interno dell'interfaccia grafica

  Vengono riportate nelle *Figure 3.1–5* le configurazioni di test considerate, i cui relativi file sono inclusi nella cartella `test/resources` del progetto.
- Per il *salvataggio* e il *caricamento* di un automa da file, sono stati considerati casi di test relativi a:
  - Lettura dell'automa da un file esistente



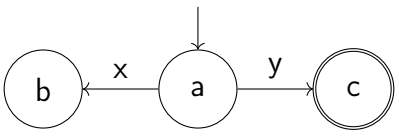
Parola di test: `xxxy`
Esito atteso: `true`
Token attesi: `{xxx, y}`
Stati attesi: `{a, b, e}`

Figura 3.1: Configurazione di test `deterministic.automata`



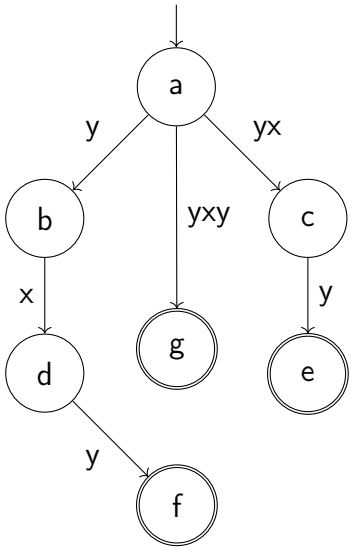
Parola di test: `xy`
Esito atteso: `true`
Token attesi: `{x, y}`
Stati attesi: `{a, c, d}`

Figura 3.2: Configurazione di test `alternative.automata`



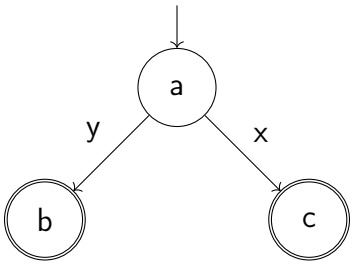
Parola di test: `x`
Esito atteso: `false`
Token attesi: `{}`
Stati attesi: `{}`

Figura 3.3: Configurazione di test `notfinal.automata`



Parola di test: `yxy`
Esito atteso: `true`
Token attesi: `{yxy}`
Stati attesi: `{a, g}`

Figura 3.4: Configurazione di test `stats.automata`



Parola di test: `z`
Esito atteso: `false`
Token attesi: `{}`
Stati attesi: `{}`

Figura 3.5: Configurazione di test `noexit.automata`

- Salvataggio dell'automa in un file non esistente
- Modifica dell'automa e successivo salvataggio in un file, per verificare che il file sia stato sovrascritto correttamente

Questi test sono stati ripetuti per ciascuna delle configurazioni di test sopra riportate.

Tutti i test, inoltre, sono stati eseguiti periodicamente, generalmente dopo rilevanti modifiche al codice sorgente, in modo da verificare che i cambiamenti apportati non avessero introdotto errori o malfunzionamenti.

3.2 Test dinamici

3.2.1 Test interni

La parte di test dinamici svolta internamente al team di sviluppo ha avuto lo scopo di verificare l'usabilità dell'interfaccia grafica realizzata e la correttezza delle azioni implementate. In seguito all'implementazione di nuove funzionalità, queste sono state sottoposte a test manuali per verificarne il corretto funzionamento.

Le categorie di test eseguite hanno così potuto spaziare su tutte le funzionalità grafiche del programma, andando a coprire casi di utilizzo non facilmente verificabili con test automatizzati. Si fa riferimento ad abitudini di utilizzo tipiche di utenti esperti, con velocità di interazione più elevate data da una conoscenza più approfondita del programma, ma anche da azioni casuali o errate che potrebbero essere compiute da utenti meno esperti. È stato ritenuto più conveniente, infatti, eseguire test manuali per coprire queste casistiche in modo più semplice, dato che l'emulazione di tali azioni non sarebbe stata agevolmente automatizzabile.

3.2.2 Test esterni

Anche la seconda parte della fase dei test dinamici è stata dedicata alla valutazione delle funzionalità del programma. Questa volta, i test di usabilità e di funzionalità eseguiti sono stati rivolti a utenti assimilabili come *finali*, con l'obiettivo di verificare che il programma fosse intuitivo e facile da utilizzare da parte di qualcuno non influenzato dalla conoscenza del progetto.

Il test è stato sottoposto secondo la seguente modalità ai diversi *tester* individuati:

1. È stato loro fornito il documento contenente le specifiche del programma;
2. In aggiunta a ciò, è stata proposta una breve descrizione del problema da un punto di vista astratto, per far comprendere all'utente il contesto di utilizzo del programma nel caso avesse avuto ancora dubbi in merito;
3. È stato aperto il programma realizzato, senza illustrare le funzionalità e le modalità di utilizzo;
4. È stato chiesto all'utente di eseguire alcune operazioni, quali:
   - l'aggiunta di un nuovo stato
   - la modifica di un stato esistente
   - la rimozione di uno stato
   - l'aggiunta di una nuova transizione
   - la modifica di una transizione esistente
   - la rimozione di una transizione

- la verifica di una stringa attraverso la ricerca di un percorso

5. I membri del team di sviluppo, in qualità di osservatori, hanno annotato le azioni compiute dall'utente e le eventuali difficoltà riscontrate.

   Un'annotazione che ha effettivamente contribuito al miglioramento del programma è stata relativa alla modalità di aggiunta di una transizione tra due stati. Una possibilità, inizialmente implementata per semplicità, era di selezionare esplicitamente uno stato di partenza e uno di arrivo attraverso dei menù a tendina. Questa soluzione, sebbene funzionante, è stata ritenuta poco intuitiva da parte degli utenti, che hanno fortemente suggerito di implementare un sistema di *drag and drop* per collegare due stati.

6. Al termine del test, è stato chiesto all'utente di esprimere un *feedback* sul programma, in particolare riguardo:

   - la gradevolezza dell'interfaccia grafica
   - la facilità di utilizzo
   - la completezza delle funzionalità
   - la presenza di errori o malfunzionamenti

Vengono ora riportati i risultati ottenuti:

Utente	Conoscenze informatiche	Gradevolezza interfaccia	Facilità di utilizzo	Completezza funzionalità	Errori
1	Sì	8/10	8/10	8/10	Nessuno
2	No	7/10	7/10	9/10	Nessuno
3	Sì	10/10	8/10	9/10	Nessuno
4	No	8/10	7/10	8/10	Nessuno
5	No	7/10	6/10	8/10	Nessuno
6	No	9/10	8/10	9/10	Nessuno
7	Sì	10/10	9/10	10/10	Nessuno
Media	—	8.4/10	7.6/10	8.7/10	—

Tabella 3.1: Risultati dei test esterni

Come anticipato, al fine di ottenere un campione quanto più possibile eterogeneo di *feedback*, i destinatari di questa fase di test sono stati di due tipologie:

(a) *Utenti con conoscenze informatiche*: la tipologia di utente che ha permesso di ottenere un *feedback* più tecnico, in grado di evidenziare eventuali criticità nell'utilizzo del programma da parte di utenti esperti;

(b) *Utenti senza conoscenze informatiche*: la tipologia di utente da cui si è potuto ottenere un *feedback* più significativo, in quanto ha permesso di capire le criticità nell'utilizzo del programma da parte di utenti non esperti. Grazie a questo, infatti, sono state apportate alcune modifiche all'interfaccia grafica non emerse in fase di sviluppo o di test precedenti.