Università degli Studi di Verona

Dipartimento di Informatica

Corso di Laurea in Informatica

Ingegneria del Software

Simulatore di Reti di Petri

Studenti: LUCA QUARESIMA MATTEO DRAGO
Matricole: VR500114 VR500241

Docenti: Prof. Pietro Sala
Prof. Carlo Combi

Anno Accademico 2024 - 2025

Contents

1 Analisi dei Requisiti

Il sistema realizzato è un simulatore di Reti di Petri che implementa un rigoroso controllo degli accessi basato sui ruoli (RBAC). L'applicazione permette la definizione formale delle reti e la loro esecuzione controllata, garantendo che le transizioni vengano eseguite solo dagli attori autorizzati secondo le specifiche di progetto.

1.1 Attori del Sistema

- **Administrator:** È responsabile della modellazione. Progetta le reti definendo la topologia (posti, transizioni, archi) e supervisiona le computazioni globali, intervenendo sulle transizioni a lui riservate.
- **End User:** È l'esecutore del processo. Sottoscrive le reti disponibili e ne avvia le istanze (computazioni), interagendo con il sistema attraverso l'esecuzione (fire) delle transizioni utente.

1.2 Elenco dei Casi d'Uso

In accordo con la Sezione 6 del documento di specifica e l'implementazione dei requisiti funzionali (FR), il sistema supporta i seguenti scenari:

1. **UC0a - Register (User):** Permette la registrazione di nuovi utenti nel `UserRepository`.
2. **UC0 - Login (User/Admin):** Gestisce l'autenticazione e il caricamento della dashboard specifica (Admin o User).
3. **UC1 - Create Petri Net (Admin):** Consente la modellazione grafica della rete e la validazione strutturale (es. unicità P_{init} e P_{final}).
4. **UC2 - Manage Computations (Admin):** Permette all'admin di monitorare tutte le istanze attive e intervenire su di esse.
5. **UC3 - Subscribe to Process (User):** Gestisce l'associazione tra un utente e una rete definita.
6. **UC4 - Start Computation (User):** Avvia una nuova sessione di simulazione, inizializzando la marcatura.
7. **UC5 - Execute Transition (User):** Gestisce la logica di scatto (fire) delle transizioni abilitate.
8. **UC6 - Manage Own Computation (User):** Permette all'utente di gestire le proprie computazioni.
9. **UC7 - Delete Petri Net (Admin):** Permette all'amministratore di eliminare definitivamente una propria rete, rimuovendo a cascata le computazioni associate.

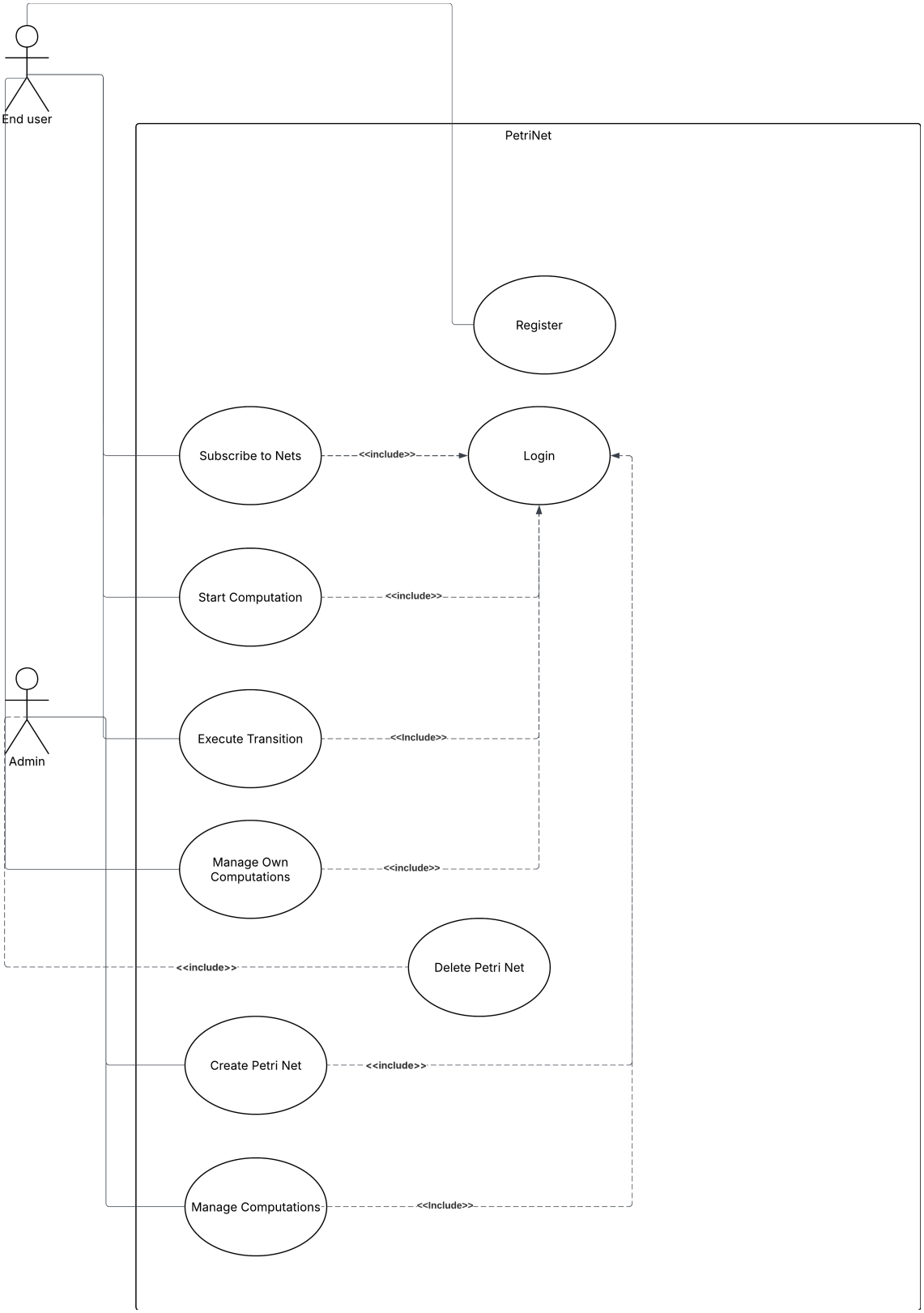

Figure 1: Diagramma dei Casi d'Uso (UCD).

2 Use-Case e Sequence Diagrams nel dettaglio

Di seguito vengono dettagliati i flussi di interazione principali. L'implementazione prevede che i Controller inoltrino le richieste al `ProcessService`, il quale interroga i Repository o aggiorna il Modello.

2.1 UC0a-UC0: Register e Login

Nome Use Case	**Register (UC0a)**
Attore	End User
Descrizione	Creazione di un nuovo account utente nel sistema.
Pre-condizioni	L'applicazione è avviata e l'utente non è loggato.
Scenario Principale	1. L'utente seleziona l'opzione di registrazione. 2. Inserisce Username e Password desiderati. 3. Il sistema verifica che l'Username non sia già in uso. 4. Il sistema salva le nuove credenziali nel `UserRepository`.
Eccezioni (Errori)	**E1. Mail Esistente:** L'utente inserisce una mail già presente nel database. Il sistema mostra un errore e richiede un altro username. **E2. Dati non validi o incompleti:** - Campi obbligatori non compilati - Formato email non valido - Password e conferma password non coincidono Il sistema segnala l'errore e richiede la correzione dei dati.
Post-condizioni	Il nuovo utente è registrato e può effettuare il login.

Nome Use Case	**Login (UC0)**
Attore	Administrator, End User
Descrizione	Procedura di autenticazione per accedere al sistema.
Pre-condizioni	L'applicazione è avviata.
Scenario Principale	1. L'Attore inserisce Username e Password. 2. Il sistema verifica le credenziali nel repository. 3. Se corrette, il sistema carica la Dashboard specifica per il ruolo (Admin o User).
Eccezioni (Errori)	**E1. Credenziali Errate:** Username o password non corrispondono. Il sistema mostra un messaggio di errore e permette di riprovare.
Post-condizioni	L'Attore è autenticato e la sessione è attiva.

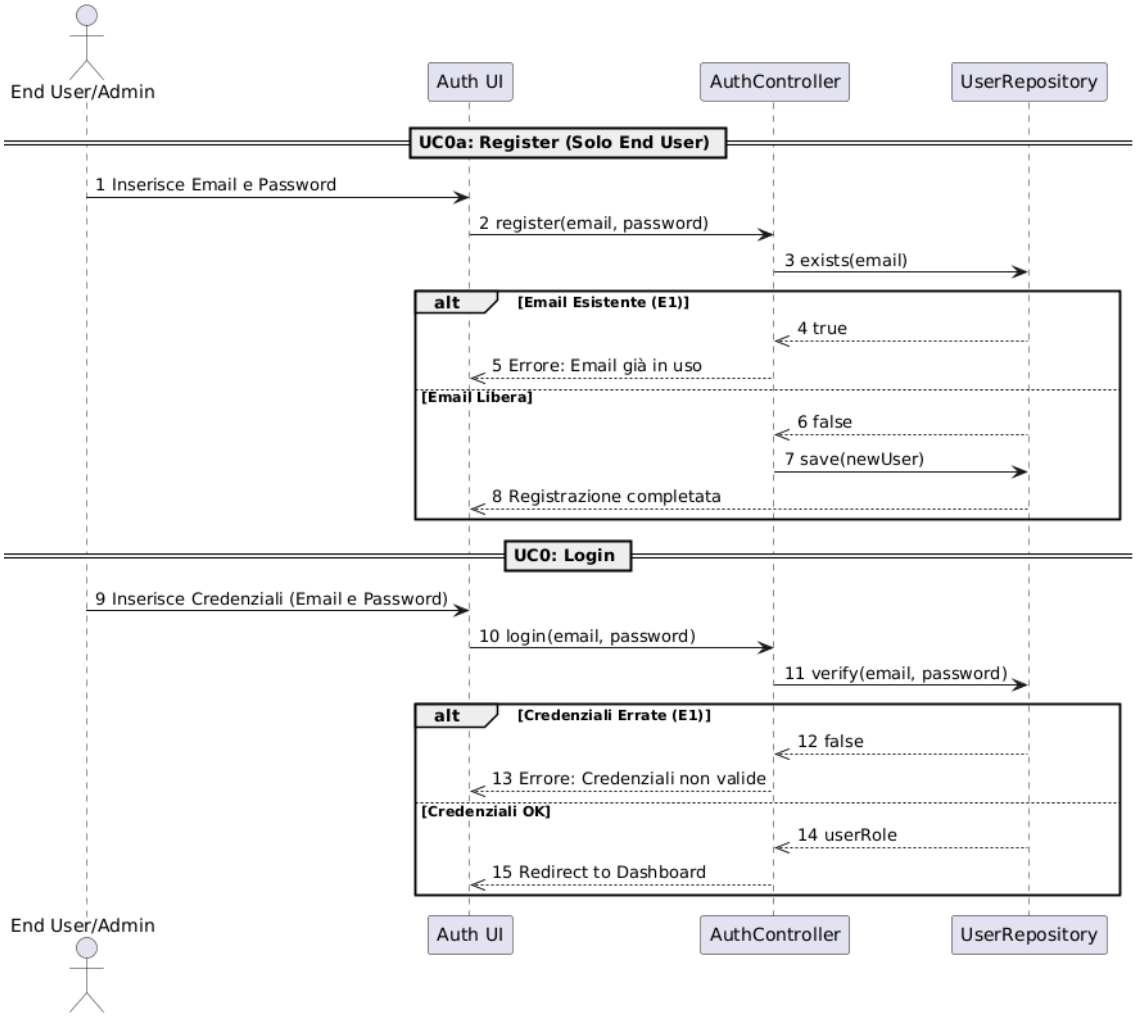

Figure 2: Sequence Diagram: Autenticazione e Registrazione.

2.2 UC1: Create Petri Net

Nome Use Case	**Create Petri Net (UC1)**
Attore	Administrator
Descrizione	L'amministratore crea una nuova rete definendo la struttura formale (posti, transizioni, archi).
Pre-condizioni	L'amministratore è autenticato.
Scenario Principale	1. L'Admin definisce l'insieme dei posti e delle transizioni. 2. L'Admin collega i nodi tramite archi orientati. 3. L'Admin designa un unico Posto Iniziale (p_{init}) e un unico Posto Finale (p_{final}). 4. L'Admin partiziona le transizioni assegnandole al ruolo "Admin" o "User". 5. Il sistema valida la struttura del grafo. 6. Il sistema salva la rete.
Eccezioni (Errori)	**E1. Violazione Strutturale:** Il sistema rileva che il Posto Iniziale ha archi in entrata o il Posto Finale ha archi in uscita. Il salvataggio è bloccato. **E2. Grafo Invalido:** La rete non è connessa, è connessa in modo errato (archi invertiti) o mancano gli elementi minimi.
Post-condizioni	La nuova rete di Petri è memorizzata e disponibile per la sottoscrizione.

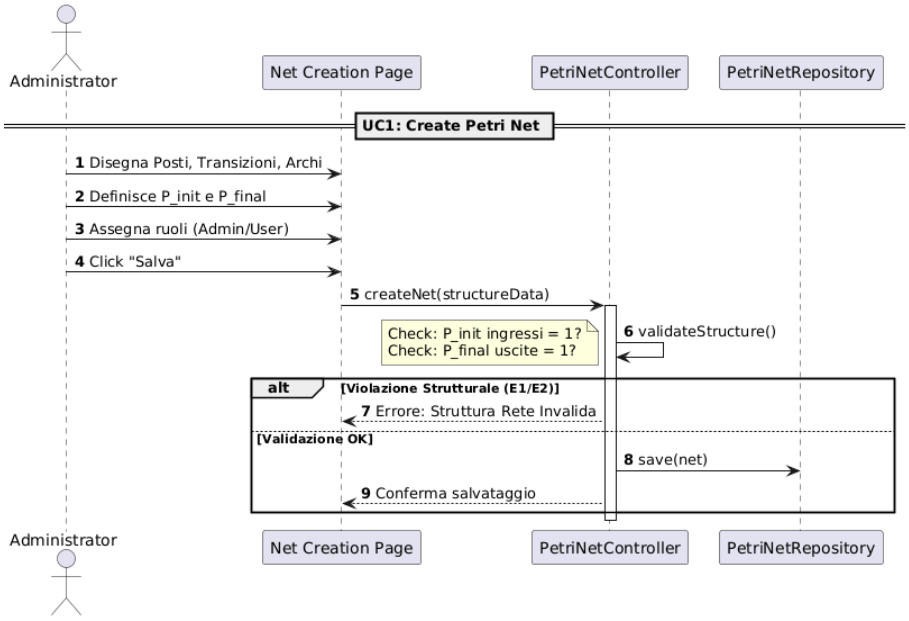

Figure 3: Sequence Diagram: Creazione e validazione della Rete.

2.3 UC2: Manage Computations

Nome Use Case	**Manage Computations (UC2)**
Attore	Administrator
Descrizione	L'amministratore monitora le computazioni attive, esegue transizioni di tipo ADMIN o elimina le istanze.
Pre-condizioni	L'amministratore è autenticato e ha creato almeno una rete con computazioni attive.
Scenario Principale	1. Il sistema mostra l'elenco delle computazioni attive sulle reti dell'Admin. 2. L'Admin seleziona una computazione specifica. 3. Il sistema visualizza lo stato corrente della rete (marcatura) e lo storico delle transizioni. 4. L'Admin analizza lo stato del processo (Monitoraggio).
Scenari Alternativi	**4a. Esecuzione Transizione (Admin):** Se sono presenti transizioni abilitate di tipo "Admin", l'amministratore le seleziona ed esegue il "fire". Il sistema aggiorna la marcatura (vedi UC5). **4b. Eliminazione:** L'Admin preme "Elimina". Il sistema rimuove la computazione.
Post-condizioni	Lo stato della computazione è aggiornato (se eseguita transizione) oppure la computazione è rimossa (se eliminata).

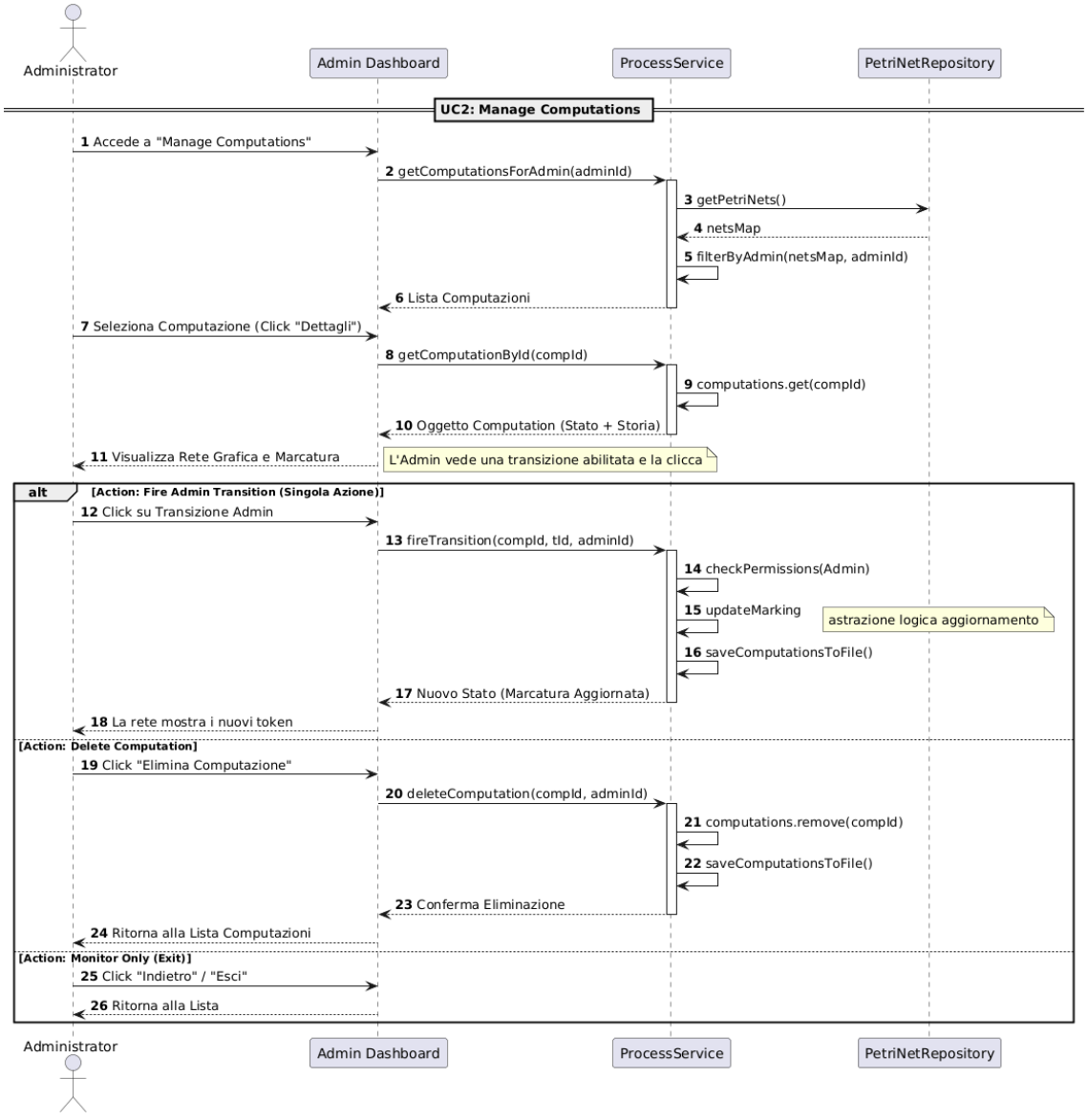

Figure 4: Sequence Diagram: Gestione computazioni lato Admin.

2.4 UC3: Subscribe to Process

Nome Use Case	**Subscribe to Process (UC3)**
Attore	End User
Descrizione	L'utente si iscrive a una rete esistente per avviare una nuova computazione. L'operazione è permessa solo se l'utente **non ha già una computazione in corso** (stato "Active") su quella stessa rete.
Pre-condizioni	L'utente è autenticato.
Scenario Principale	1. Il sistema mostra la lista delle reti Petri disponibili. 2. L'utente clicca per avviare una nuova computazione su una rete desiderata. 3. Il sistema verifica che l'utente non abbia computazioni "Active" per quella specifica rete. 4. Il sistema crea una nuova istanza di computazione (inizializzando la marcatura e impostando lo stato su Active). 5. **Il sistema reindirizza automaticamente l'utente** alla pagina della computazione appena creata.
Eccezioni (Errori)	**E1. Computazione Già Attiva:** Se l'utente ha già una computazione in stato "Active" sulla rete selezionata, il sistema blocca l'operazione e mostra un errore. Eventuali computazioni in stato "Completed" o "Unknown" non bloccano la nuova sottoscrizione.
Post-condizioni	Una nuova computazione "Active" viene creata e l'utente viene reindirizzato alla vista interattiva per iniziare a operare.

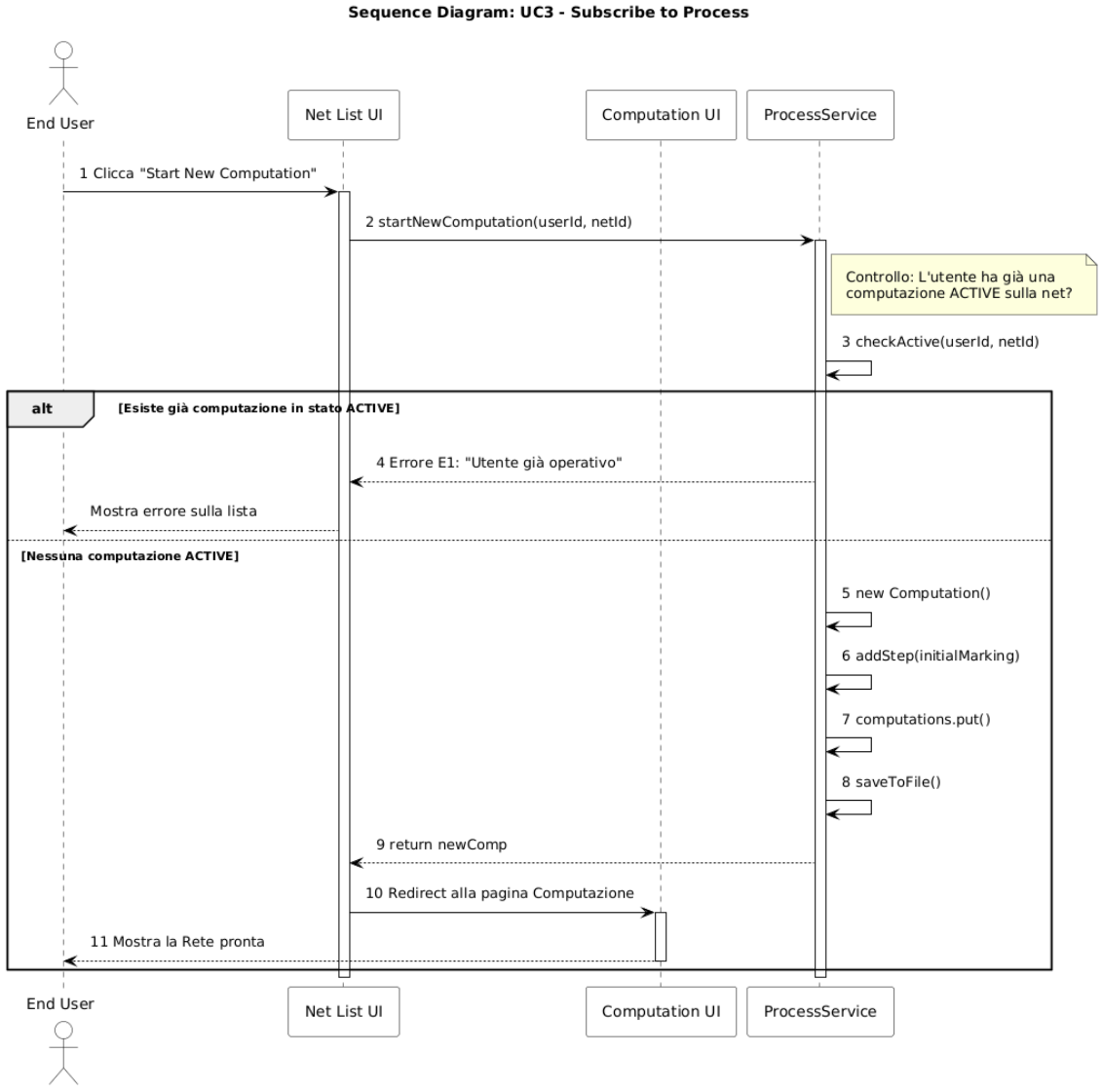

Figure 5: Sequence Diagram: Sottoscrizione utente.

2.5 UC4: Continue Computation

Nome Use Case	**Computation (UC4)**
Attore	End User
Descrizione	L'utente continua una computazione attiva
Pre-condizioni	L'utente è sottoscritto alla rete e **NON deve esistere nessun'altra computazione in stato "Active" per questo utente su questa rete.**
Scenario Principale	1. L'utente può trovarsi nella pagina di dettaglio della rete (da UC3 o navigazione attraverso View Details). 2. Il sistema verifica che non ci siano altre istanze attive per l'utente sulla stessa rete. 3. Il sistema mostra la rete con la marcatura corrente. 4. L'utente interagisce con la net
Post-condizioni	**P1. Computazione non terminata:** Il sistema rileva che la computazione non è terminata, in caso di uscita sarà possibile ri-entrare. **P2. Computazione terminata:** Il sistema rileva che la computazione è terminata. Quando si esce si potrà comunque rientrare, ma in view only.

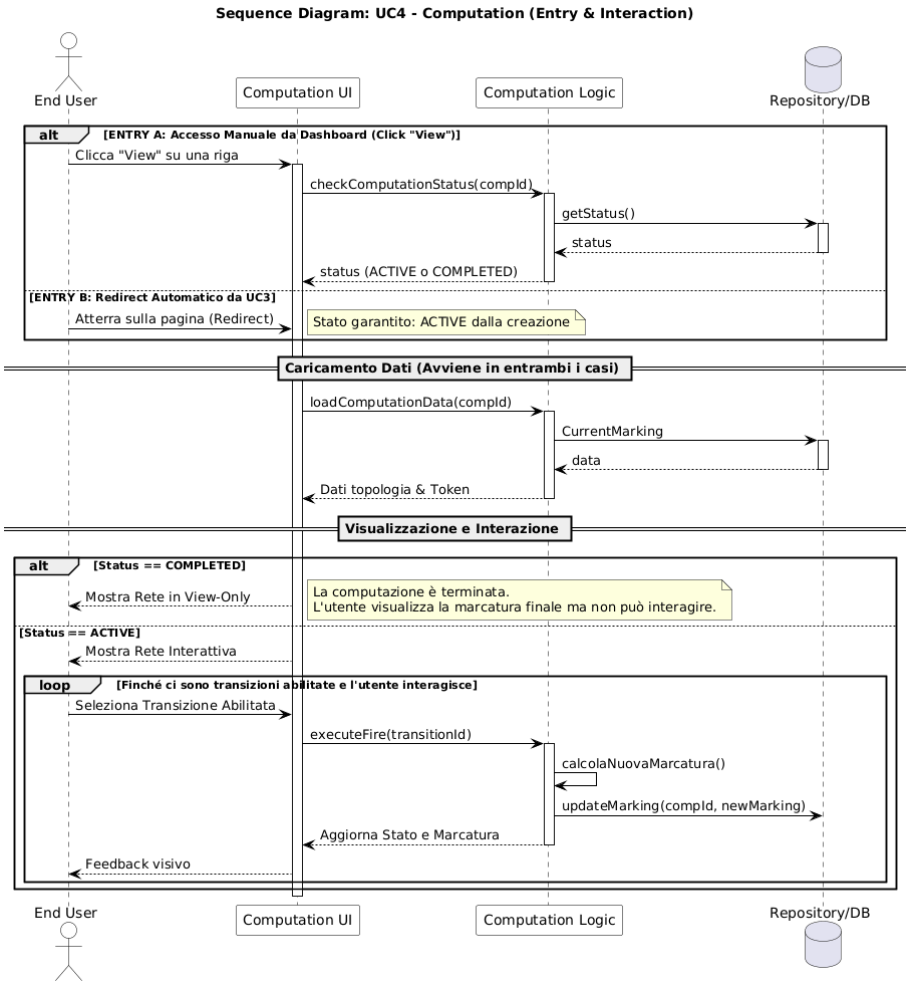

Figure 6: Sequence Diagram: Continuazione o avvio computazione.

2.6 UC5: Execute Transition

Nome Use Case	**Execute Transition (UC5)**
Attore	End User, Administrator
Descrizione	L'attore esegue una transizione abilitata coerente con il proprio ruolo.
Pre-condizioni	Computazione attiva. Esiste almeno una transizione abilitata (token nei posti di input).
Scenario Principale	1. Il sistema calcola le transizioni abilitate basandosi sulla marcatura corrente. 2. Il sistema evidenzia le transizioni in base al ruolo (Admin o User). 3. L'Attore clicca la transizione da eseguire ("Fire"). 4. Il sistema rimuove i token dai posti di input e li aggiunge ai posti di output.
Scenari Alternativi	**4a. Completamento Processo:** Se un token viene posizionato nel Posto Finale (p_{final}), il sistema marca la computazione come "Completed".
Eccezioni (Errori)	**E1. Violazione Ruolo:** L'attore tenta di eseguire una transizione riservata all'altro ruolo (es. User prova a eseguire una transizione Admin). **E2. Transizione non Abilitata:** I token non sono sufficienti per attivare la transizione.
Post-condizioni	La marcatura della rete è aggiornata.

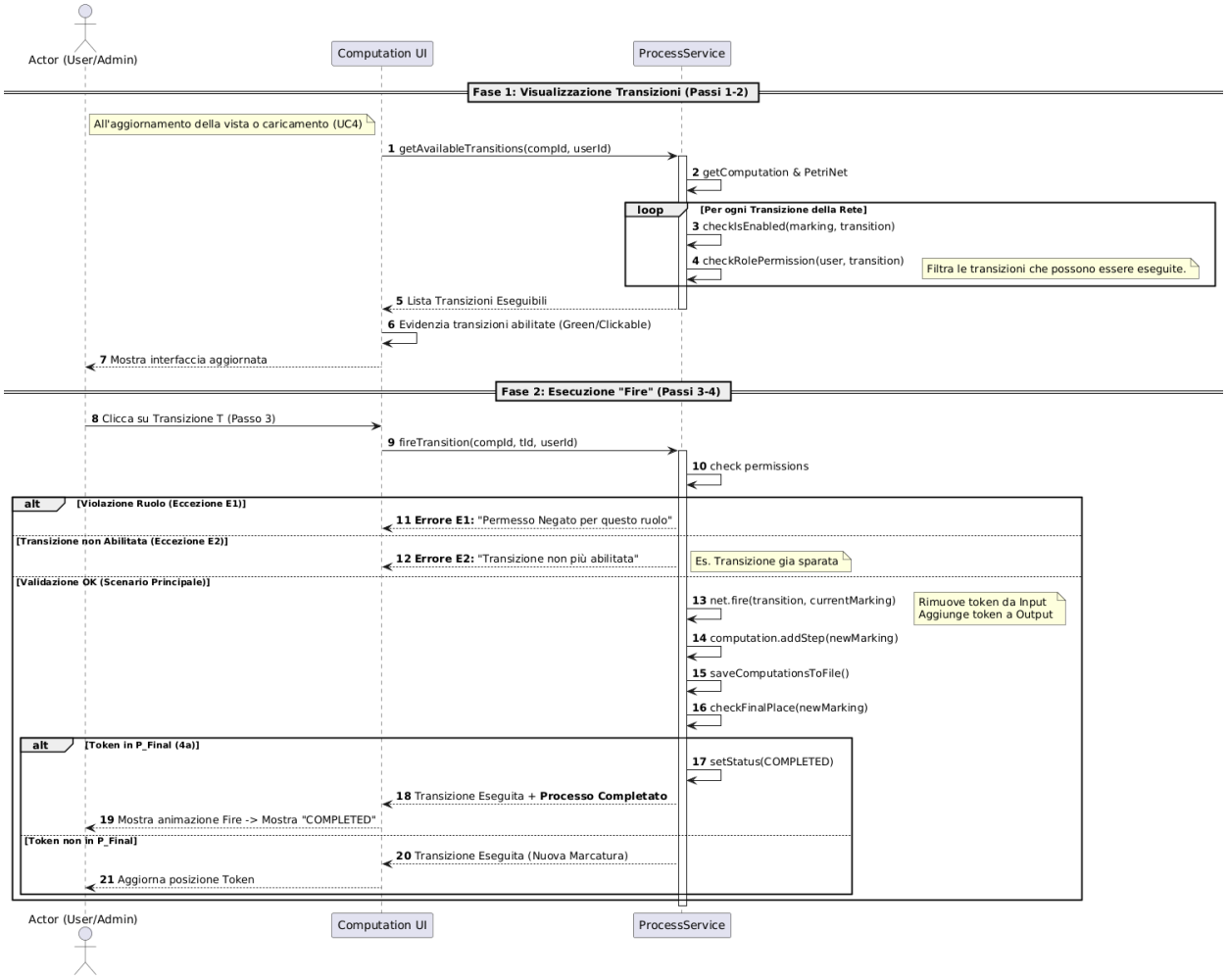

Figure 7: Sequence Diagram: Esecuzione di una transizione.

2.7 UC6: Delete Own Computations

Nome Use Case	**Delete Own Computations (UC6)**
Attore	End User
Descrizione	L'utente visualizza le net a cui è iscritto e può eliminare le proprie computazioni.
Pre-condizioni	L'utente possiede computazioni (attive o completate).
Scenario Principale	1. L'utente seleziona la net da eliminare. 2. L'utente clicca su "Elimina" per cancellare i dati. 3. Il sistema rimuove la computazione dal database.
Scenari Alternativi	**1a. Filtro:** L'utente filtra le computazioni per stato (Active/Completed).
Post-condizioni	La computazione è rimossa o visualizzata.

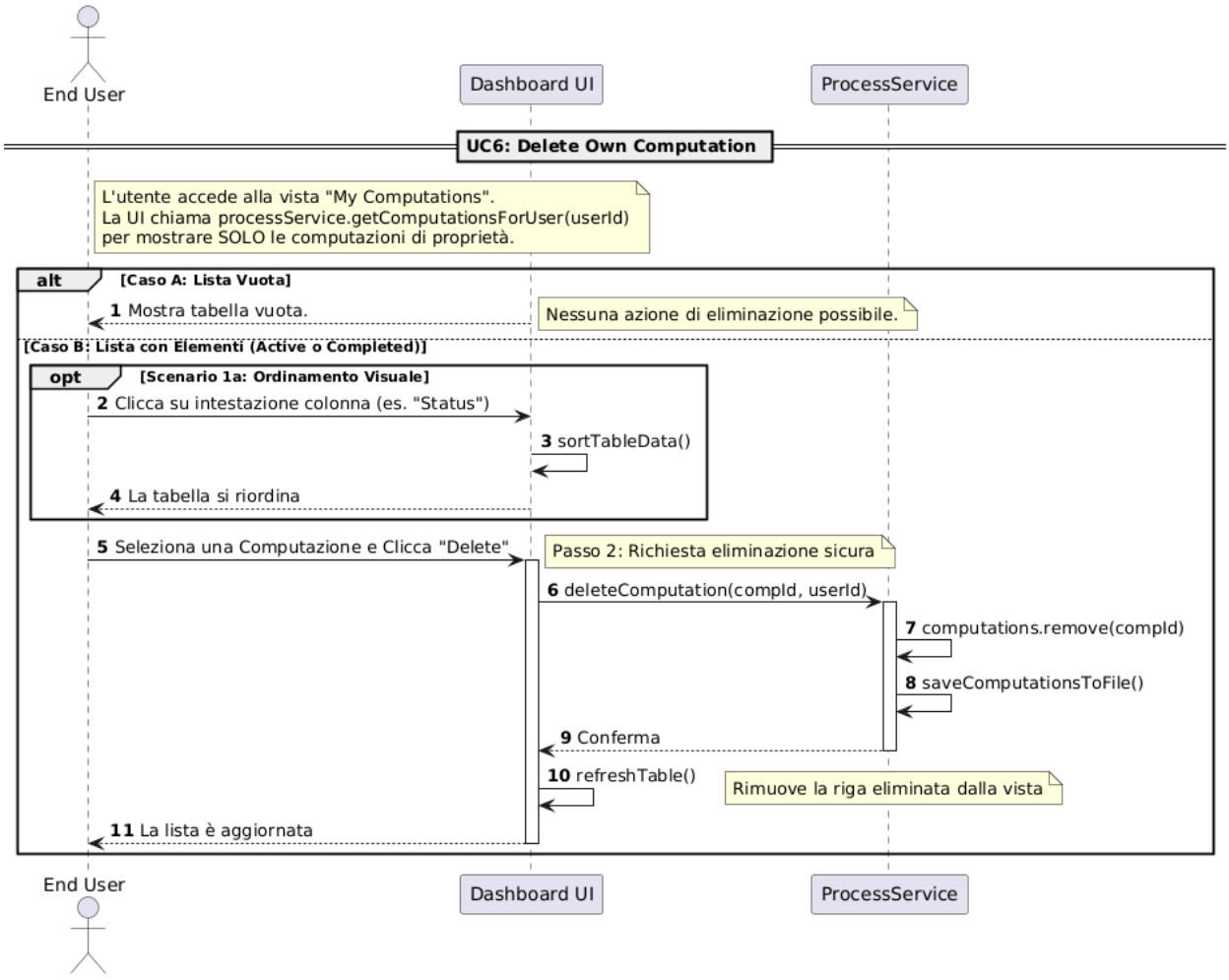

Figure 8: Sequence Diagram: Cancellazione computazione.

2.8 UC7: Delete Petri Net

Nome Use Case	**Delete Petri Net (UC7)**
Attore	Administrator
Descrizione	L'amministratore elimina definitivamente una propria Rete di Petri dal sistema. L'operazione è consentita **solo se non vi sono computazioni in stato "Active"** associate alla rete. Le computazioni già terminate (storiche) **non** vengono eliminate a cascata, ma rimangono accessibili agli utenti (diventando orfane e visualizzate come "Unknown").
Pre-condizioni	L'amministratore è autenticato. L'interfaccia grafica è filtrata per mostrare esclusivamente le reti di cui l'Admin è proprietario.
Scenario Principale	1. L'Admin visualizza l'elenco delle proprie reti nella Dashboard. 2. Clicca sul pulsante "Elimina" in corrispondenza della rete desiderata. 3. Il sistema verifica che **nessun utente** abbia una computazione in stato "Active" su quella specifica rete. 4. Il sistema mostra un pop-up che richiede la conferma definitiva dell'eliminazione, avvisando che le computazioni terminate rimarranno visibili agli utenti come "Unknown". 5. L'Admin conferma l'operazione. 6. La rete viene rimossa definitivamente dal `PetriNetRepository`. 7. I file JSON di persistenza vengono aggiornati.
Eccezioni (Flussi Alternativi)	**E1. Computazioni Attive Presenti:** Al passo 3, se il sistema rileva una o più computazioni "Active", l'eliminazione viene bloccata per mantenere la consistenza in esecuzione. Viene mostrato un messaggio di errore. **E2. Annullamento:** Al passo 5, se l'Admin clicca su "Annulla" nel pop-up, l'operazione si interrompe senza alterare i dati.
Post-condizioni	La Rete di Petri è rimossa dal repository. Le computazioni in stato "Completed" associate ad essa rimangono salvate per gli utenti, ma se aperte verranno mostrate in sola lettura (View-Only) con il nome della rete indicato come "Unknown".

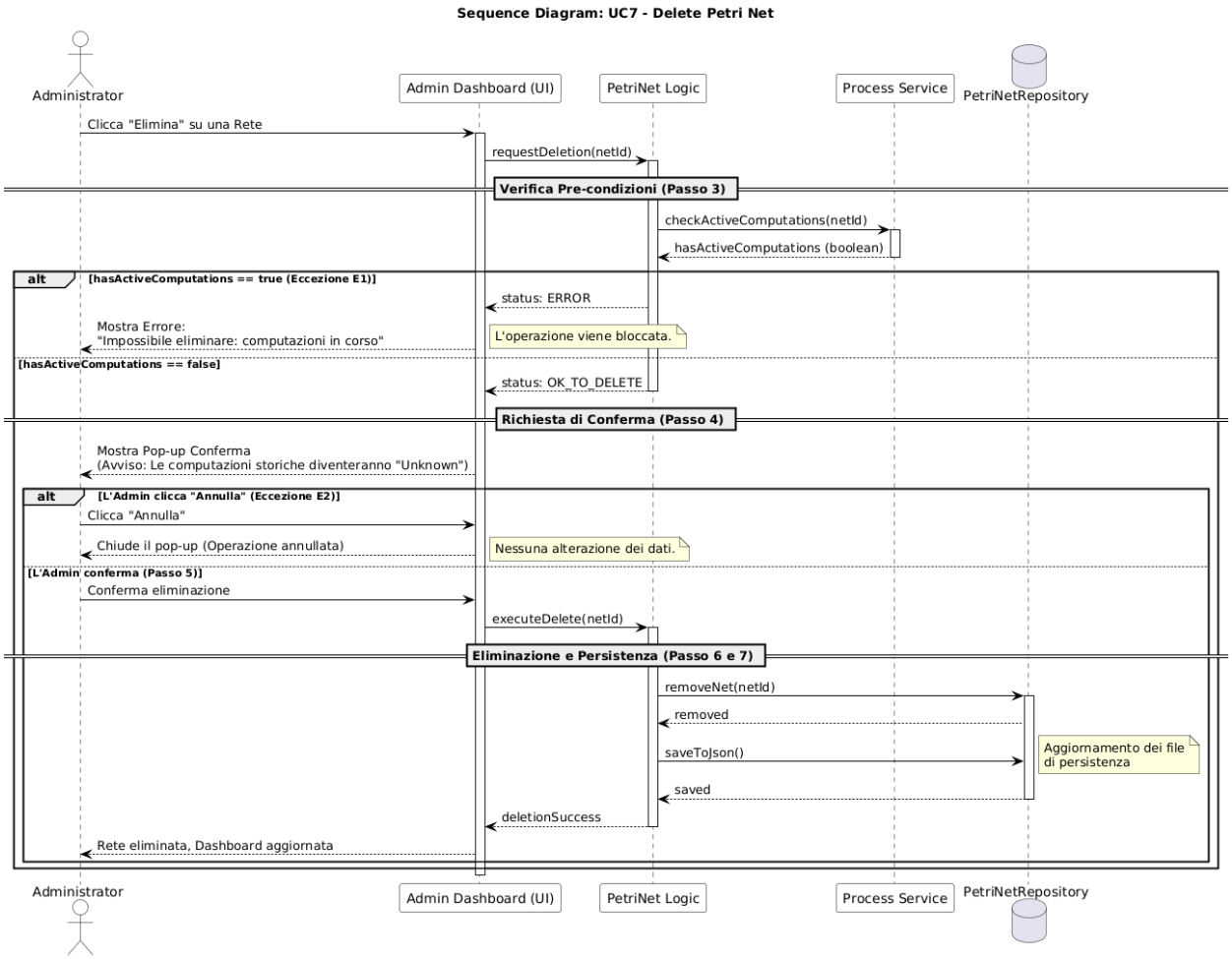

Figure 9: Sequence Diagram: Cancellazione rete.

3 Diagrammi di attività

I diagrammi di attività sono stati strutturati per rispecchiare i flussi operativi implementati nel codice. Si distinguono tre macro-aree: l'autenticazione (gestita dai controller comuni), le operazioni dell'Admin e il ciclo di vita dell'Utente (simulazione).

3.1 Accesso al Sistema

Il primo diagramma illustra il processo di *dispatching* gestito all'avvio: il sistema verifica le credenziali tramite il `UserRepository` e indirizza l'utente alla dashboard corretta in base al ruolo recuperato (Admin o User).

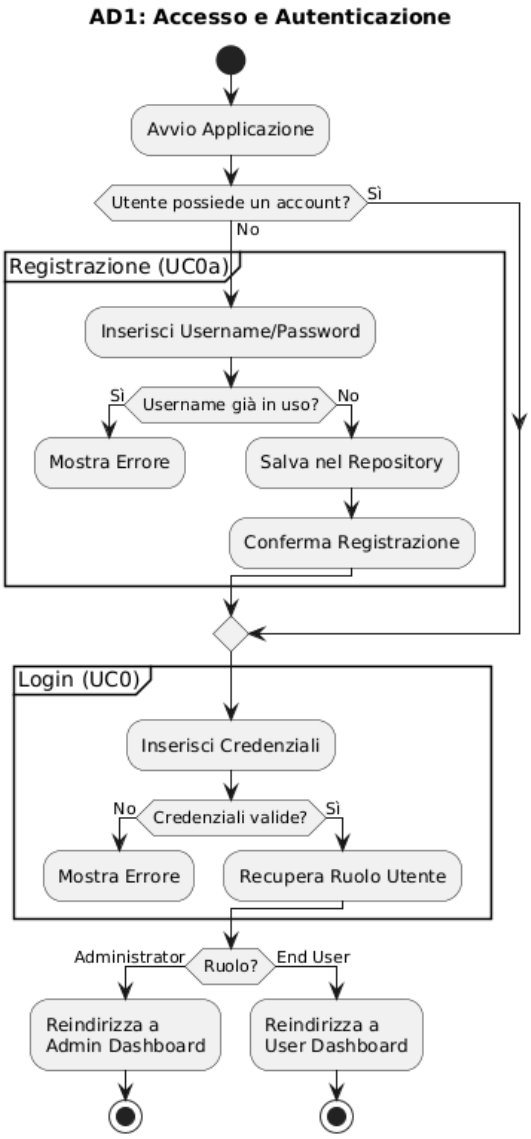

Figure 10: Activity Diagram: Flusso di Registrazione e Login.

3.2 Workflow Amministratore

Il diagramma seguente copre le responsabilità dell'Admin implementate in `AdminAreaController`. Sono evidenti due rami paralleli:

- **Progettazione:** Creazione e salvataggio della topologia della rete.
- **Monitoraggio:** Supervisione delle computazioni attive con possibilità di intervento tramite transizioni privilegiate.

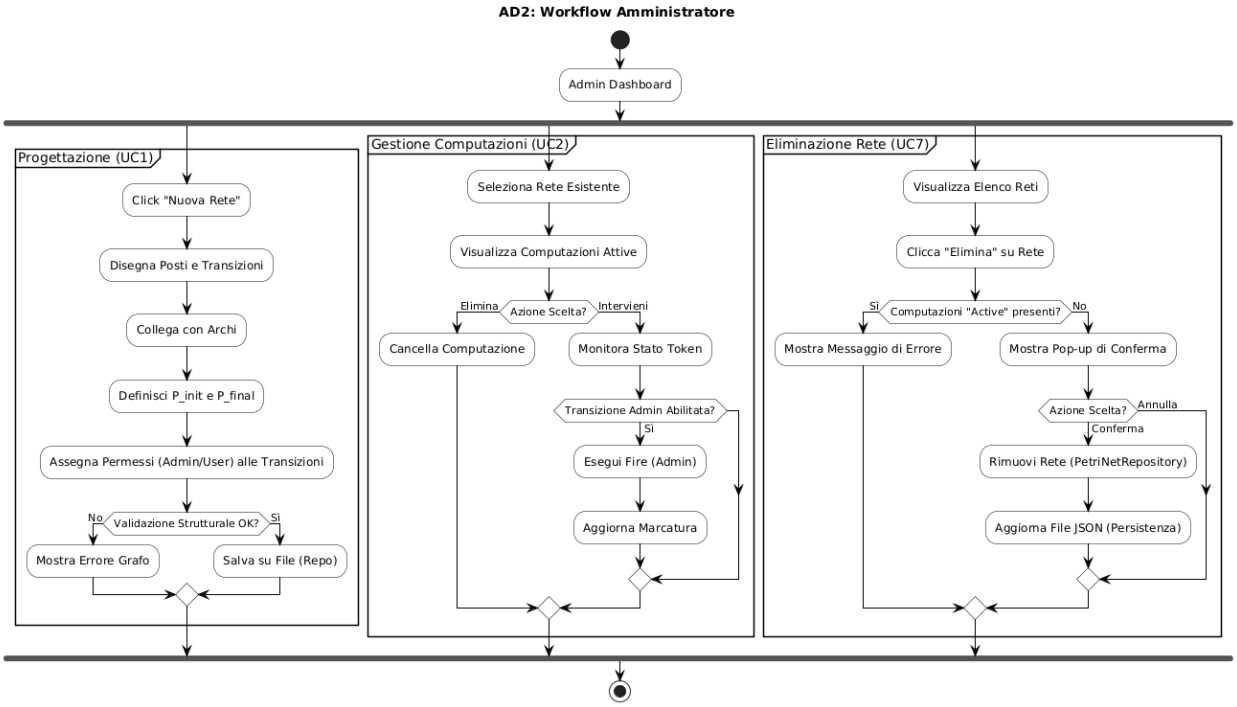

Figure 11: Activity Diagram: Creazione Rete e Gestione Computazioni.

3.3 Workflow Utente (Esecuzione)

Il diagramma di esecuzione mostra il ciclo principale gestito dal `ProcessService`. L'utente visualizza lo stato corrente, sceglie tra le transizioni abilitate e attende che il sistema elabori il nuovo stato fino al raggiungimento della condizione di terminazione (P_{final}).

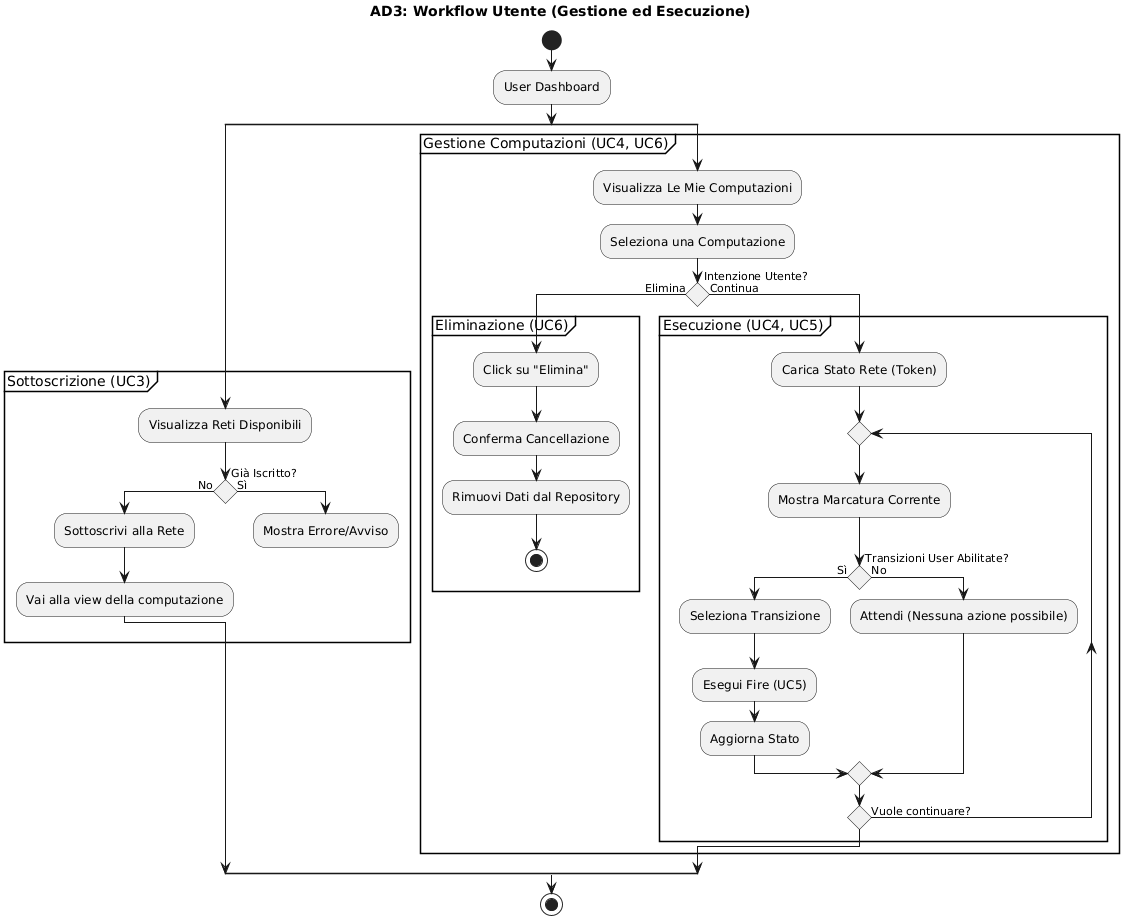

Figure 12: Activity Diagram: Ciclo di esecuzione della Rete di Petri.

4 Metodologia di Sviluppo e Gestione del Progetto

Per la realizzazione del simulatore, è stata adottata una metodologia di **sviluppo Iterativo e Incrementale**, ispirata ai principi del framework **Agile**. Questo approccio si è rivelato la scelta ottimale, garantendo flessibilità e rapidità di adattamento ai requisiti.
Lo sviluppo del sistema è partito dall'analisi delle specifiche di progetto, con l'obiettivo di trasformare il modello matematico delle Reti di Petri in un'applicazione software interattiva e sicura.

Dallo studio del documento iniziale ci siamo concentrati su tre aspetti fondamentali:

- **Regole del Dominio:** Implementazione rigorosa delle logiche di rete (regole di abilitazione e scatto) e corretta gestione dei posti iniziale (p_{init}) e finale (p_{final}) per definire il ciclo di vita dei processi.
- **Gestione dei Ruoli (RBAC):** Separazione netta delle responsabilità tra Amministratore (creazione delle reti e supervisione) e Utente (sottoscrizione ed esecuzione).
- **Estensione dei Requisiti:** Come suggerito dalle specifiche, abbiamo ampliato le funzionalità base, implementando un'interfaccia grafica per la visualizzazione e la gestione delle reti.

4.1 Ciclo di Sviluppo Incrementale

Il software non è stato sviluppato in un'unica fase, ma è stato suddiviso in *incrementi* funzionali. Ogni iterazione ha previsto la progettazione, l'implementazione e la verifica di una specifica componente. Ad esempio:

1. **Logica:** Definizione del modello matematico della Rete di Petri (`Place`, `Transition`, `Arc`).
2. **Motore e Architettura:** Sviluppo del `ProcessService` e implementazione del pattern Facade e Strategy.
3. **Interfaccia e Persistenza:** Integrazione grafica tramite JavaFX (Observer e Factory pattern) e collegamento ai file JSON.

4.2 Strumenti di Collaborazione e Versioning

Data la natura distribuita del lavoro, il coordinamento è stato gestito attraverso

- **GitHub:** Utilizzato come repository centralizzato per la gestione del codice sorgente. Git ha permesso di lavorare in parallelo sulle diverse funzionalità, gestendo i conflitti e mantenendo uno storico sicuro delle modifiche e dei progressi (tramite *commit* continui).

4.3 Approccio al Testing

Non essendo stata adottata rigorosamente una metodologia *Test-Driven Development* fin dall'inizio, la verifica della qualità del software ha seguito un approccio ibrido. Per quanto riguarda l'interfaccia utente, è stato condotto un **Testing Manuale**. Questo ha permesso di verificare immediatamente il comportamento visivo dei nuovi blocchi di codice, come il corretto scatto di una transizione o il posizionamento degli archi. Nella fase di consolidamento, per il motore di calcolo e le regole di business è stata sviluppata una serie di **Test Automatizzati**.

5 Pattern Architetturali

L'applicazione segue il pattern **MVC (Model-View-Controller)**, supportato da un **Service Layer** centrale (`ProcessService`) e da componenti per la configurazione (`SharedResources`).

5.1 Pattern Architetturale: MVC e Service Layer

L'organizzazione delle classi rispetta la seguente suddivisione:

1. **Model (Dominio):** Le classi come `PetriNet`, `Place`, `Transition` e `Computation` rappresentano le entità pure del dominio. Sono prive di logica grafica e gestiscono solo la propria consistenza interna.

2. **View (Interfaccia):** Definita nei file FXML, si occupa del layout. I componenti grafici sono aggiornati dinamicamente tramite **binding e listener**.

3. **Controller:** Classi come `AdminAreaController` e `ViewPetriNetController` gestiscono gli eventi utente. Non contengono logica di business complessa, ma delegano le operazioni al Service Layer.

Per evitare la duplicazione della logica, è stato introdotto il **ProcessService**. Questa classe funge da facciata per tutte le operazioni critiche (avvio computazione, scatto transizioni, eliminazione), garantendo che le regole di business siano applicate uniformemente.

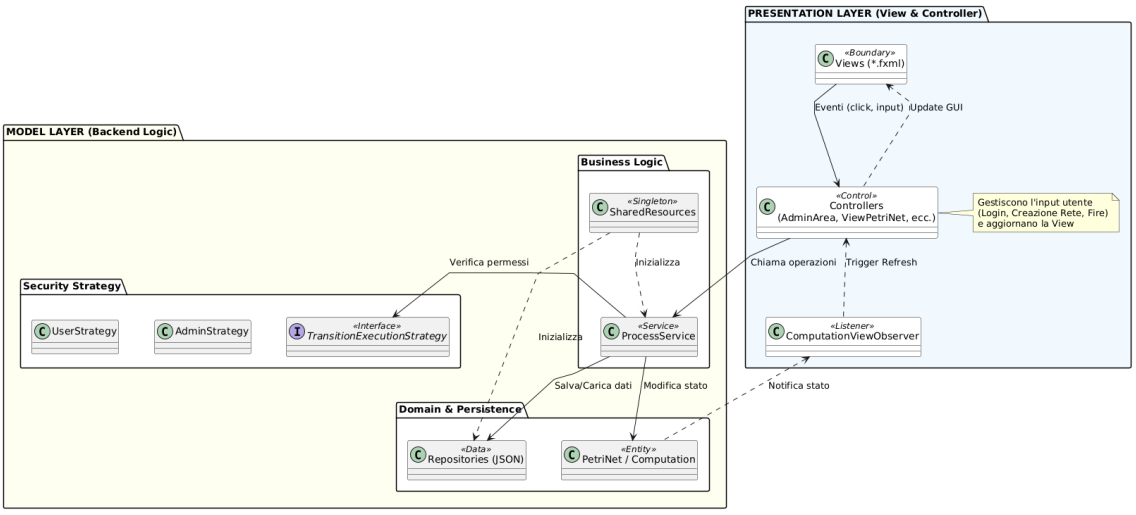

Figure 13: MVC Diagram.

5.2 Design Pattern Applicati

Sono stati implementati i seguenti pattern Go4:

5.2.1 Singleton Pattern

La classe `SharedResources` è stata implementata come **Singleton**. Essa viene istanziata una sola volta all'avvio e si occupa di creare e detenere le istanze uniche di `UserRepository`, `PetriNetRepository` e `ProcessService`. Questo garantisce che tutti i controller accedano allo stesso stato condiviso.

```java
package application.logic;

import application.repositories.PetriNetRepository;
import application.repositories.UserRepository;

/**
 * Provides global access to repositories.
 */
public class SharedResources {

    // 1. Istanza statica e final
    private static final SharedResources instance = new SharedResources();

    private final UserRepository userRepository;
    private final PetriNetRepository petriNetRepository;
    private final ProcessService processService;

    // 2. Costruttore privato
    private SharedResources() {
        // Instantiated only once at the start of the application.
        this.userRepository = new UserRepository();
        this.petriNetRepository = new PetriNetRepository();
        this.processService = new ProcessService(userRepository, petriNetRepository);
    }

    /**
     * 3. Punto di accesso globale all'unica istanza.
     */
    public static SharedResources getInstance() {
        return instance;
    }
}
```

Listing 1: Implementazione del Singleton Pattern: `SharedResources`

5.2.2 Strategy Pattern (RBAC)

Per gestire il requisito funzionale che limita l'esecuzione delle transizioni in base al ruolo, è stato applicato il pattern **Strategy**.
Il metodo `fireTransition` del servizio delega il controllo dei permessi all'interfaccia `TransitionExecutionStrategy`. A runtime, viene istanziata la strategia concreta corretta (`AdminExecutionStrategy` o `UserExecutionStrategy`) in base al tipo della transizione selezionata.

Di seguito si riportano l'interfaccia base e le due strategie concrete:

```
// 1. L'Interfaccia (The Strategy)
public interface TransitionExecutionStrategy {
    void checkPermissions(User user, PetriNet net, Transition transition);
}

// 2. Strategia Concreta per le transizioni USER
public class UserExecutionStrategy implements TransitionExecutionStrategy {
    @Override
    public void checkPermissions(User user, PetriNet net, Transition transition) {
        boolean isNetAdmin = user.isAdmin() && net.getAdminId().equals(user.getId());

        if (isNetAdmin) {
            throw new IllegalStateException(
                "L'Admin non puo' eseguire transizioni utente sulla propria rete."
            );
        }
    }
}

// 3. Strategia Concreta per le transizioni ADMIN
public class AdminExecutionStrategy implements TransitionExecutionStrategy {
    @Override
    public void checkPermissions(User user, PetriNet net, Transition transition) {
        boolean isNetAdmin = user.isAdmin() && net.getAdminId().equals(user.getId());

        if (!isNetAdmin) {
            throw new IllegalStateException(
                "Non sei autorizzato a eseguire transizioni ADMIN."
            );
        }
    }
}
```

Listing 2: Implementazione dello Strategy Pattern: Interfaccia e Strategie Concrete

5.2.3 Observer Pattern (Sincronizzazione UI)

Per mantenere l'interfaccia grafica sincronizzata con l'evoluzione della computazione, è stato utilizzato il pattern **Observer**. La classe `Computation` notifica i cambiamenti di stato agli osservatori registrati. La classe `ComputationViewObserver` riceve queste notifiche e invoca il metodo `refreshState()` sul controller della vista, garantendo un disaccoppiamento completo tra il motore di esecuzione e la GUI.

Il "ponte" tra la logica e la grafica viene creato all'interno del Controller. L'iscrizione avviene durante il caricamento della computazione:

```
// All'interno di ComputationObserver.java
public abstract class ComputationObserver {
    public abstract void update(Computation updatedComputation);
}

// All'interno di Computation.java (Subject)
public class Computation {
    private transient List<ComputationObserver> observers = new
    ArrayList<>();

    public void attach(ComputationObserver observer) {
        if(observers==null) observers = new ArrayList<>();
        observers.add(observer);
    }

    private void notifyObservers() {
        if (observers == null) observers = new ArrayList<>();
        for (ComputationObserver observer : observers) {
            observer.update(this);
        }
    }

    public void addStep(ComputationStep step) {
        steps.add(step);
        notifyObservers();
    }
}

// All'interno di ViewPetriNetController.java
public void loadComputation(User user, Computation computation) {
    // Viene creato il Concrete Observer che si iscrive al Subject
    (Computation)
    this.viewObserver = new ComputationViewObserver(computation,
    this);
}

// All'interno di ComputationViewObserver.java
@Override
public void update(Computation updatedComputation) {
    // Quando la rete scatta, l'Observer comanda alla UI di
    aggiornarsi
    view.refreshState();
}
```

Listing 3: Registrazione dell'Observer per aggiornare la UI

5.2.4 Facade Pattern

Per gestire la complessità delle operazioni e isolare l'interfaccia utente dalle logiche di basso livello, la classe `ProcessService` è stata progettata implementando il pattern **Facade**.

In accordo con la definizione del pattern, il `ProcessService` fornisce un'interfaccia unificata e semplificata per l'accesso a un sottosistema molto complesso. Senza questo pattern, i Controller di JavaFX (come `ViewPetriNetController`) dovrebbero gestire manualmente una vasta gamma di dipendenze ogni volta che un utente esegue un'azione (es. il "fire" di una transizione):

- L'accesso e il recupero dei dati da `UserRepository` e `PetriNetRepository`.
- La delega dei controlli di sicurezza alle interfacce `TransitionExecutionStrategy`.
- L'applicazione delle regole matematiche della rete tramite le classi di dominio (`PetriNet`, `MarkingData`).
- La creazione dello storico (`ComputationStep`) e la serializzazione su file JSON tramite `ObjectMapper` di Jackson.

Incapsulando tutte queste interazioni in metodi ad alto livello come `fireTransition(...)` e `startNewComputation(...)`, il Facade alleggerisce i Controller e garantisce il disaccoppiamento totale tra la GUI e la logica.

```
1  public class ProcessService {
2      // Sottosistemi coordinati dal Facade
3      private final UserRepository userRepository;
4      private final PetriNetRepository petriNetRepository;
5      private final ObjectMapper mapper = new ObjectMapper(); //
       Persistenza JSON
6      private final Map<String, Computation> computations = new
       HashMap<>();
7
8      /**
9       * Metodo "Facade" che nasconde la complessita' di uno scatto.
10      * Coordina: Sicurezza, Logica di Dominio e Persistenza.
11      */
12     public void fireTransition(String computationId, String
       transitionId, String userId) {
13         // 1. Recupero dei dati dai sottosistemi
14         Computation comp = computations.get(computationId);
15         User user = userRepository.getUserById(userId);
16         PetriNet net = comp.getPetriNetSnapshot();
17
18         // 2. Delega alla logica di sicurezza (Strategy Pattern)
19         checkFirePermissions(user, net, net.getTransitions().get(
       transitionId));
20
21         // 3. Esecuzione delle regole delle Petri Net
22         MarkingData newMarking = net.fire(transitionId, comp.
       getLastStep().getMarkingData());
23
24         // 4. Gestione dello stato e persistenza Jackson
```

```
25          comp.addStep(new ComputationStep(comp.getId(), transitionId
    , newMarking));
26          saveComputationsToFile(); // Nasconde i dettagli del File I
    /O
27      }
28  }
```

Listing 4: ProcessService come Facade: coordinamento di sottosistemi

5.2.5 Simple Factory Pattern (Generazione UI)

Per gestire la creazione dinamica degli elementi grafici della Rete di Petri (Posti, Transizioni e Archi) mantenendo pulito il codice dei Controller, è stato utilizzato il pattern **Simple Factory**.

Classi come `ArcViewFactory`, `PlaceViewFactory` e `TransitionViewFactory`, contengono metodi statici per la generazione dei nodi JavaFX.(`Line`, `Circle`, `Rectangle`). L'uso di queste factory ha permesso di incapsulare:

- **Lo Styling Grafico:** Colori, dimensioni e spessori di default dei nodi.
- **Il Property Binding:** La logica complessa che ancora dinamicamente le linee degli archi alle coordinate (x, y) dei nodi sorgente e destinazione.

In questo modo, la logica di inizializzazione della View è stata totalmente separata dalla gestione degli eventi del Controller.

Di seguito uno dei tre metodi implementati:

```
1  public class PlaceViewFactory {
2
3      /**
4       * Factory Method: incapsula la creazione di un Group JavaFX
5       * nascondendo la complessita' di styling e binding.
6       */
7      public static Group createPlaceNode(
8              Place place,
9              String labelText,
10             double x,
11             double y,
12             Consumer<Place> onInitial,
13             Consumer<Place> onFinal
14     ) {
15         // 1. Istanziazione e configurazione dello stile
16         Circle placeCircle = new Circle(0, 0, 20, Color.LIGHTBLUE);
17         placeCircle.setStroke(Color.BLACK);
18         Text placeLabel = new Text(labelText);
19
20         // 2. Assemblaggio del componente
21         Group placeNode = new Group(placeCircle, placeLabel);
22         placeNode.setLayoutX(x);
23         placeNode.setLayoutY(y);
24
25         // 3. Incapsulamento della logica di interazione
26         setupInteractions(placeNode, place, onInitial, onFinal);
```

```
27
28              return placeNode;
29          }
30  }
```

Listing 5: Simple Factory per la generazione dei nodi grafici

5.3 Diagrammi UML di Supporto

Di seguito sono riportati i diagrammi delle classi che dettagliano le strutture dati, la logica di esecuzione e l'architettura dei servizi descritta.

5.3.1 Modello Strutturale della Rete

La Figura 14 illustra le entità del dominio (`PetriNet`, `Place`, `Transition`). Si evidenzia l'uso di UUID per l'identificazione univoca degli elementi.

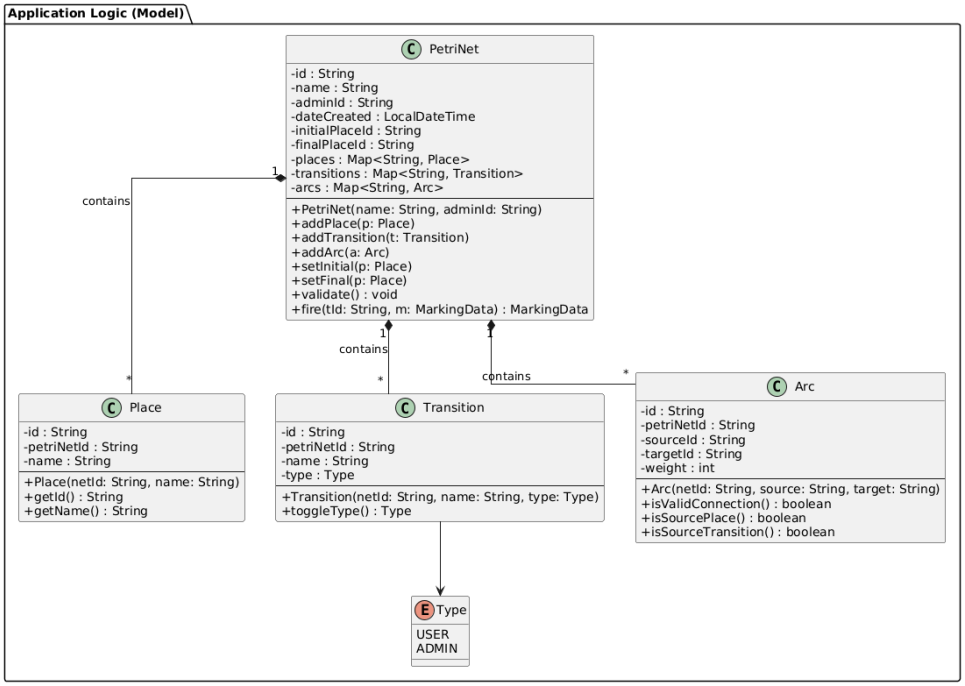

Figure 14: Class Diagram: Struttura dati del Modello.

5.3.2 Gestione dell'Esecuzione

La Figura 15 mostra le classi dedicate al runtime (`Computation`, `MarkingData`) e l'integrazione del pattern Observer tramite `ComputationViewObserver`.

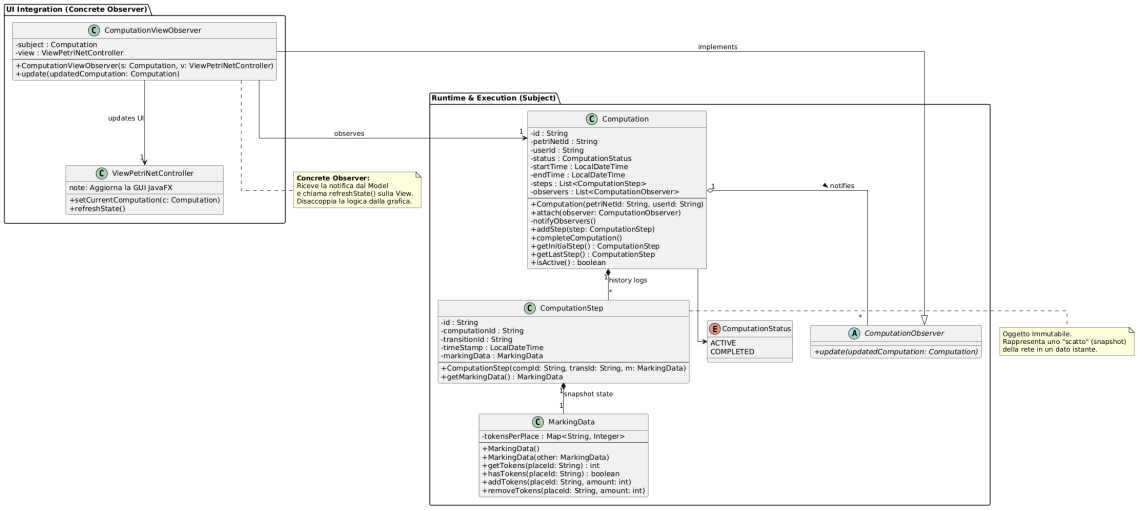

Figure 15: Class Diagram: Runtime e Pattern Observer.

5.3.3 Architettura Core e Sicurezza

La Figura 16 rappresenta il cuore del sistema: il `ProcessService`, il Singleton `SharedResources` e l'implementazione delle Strategie di sicurezza.

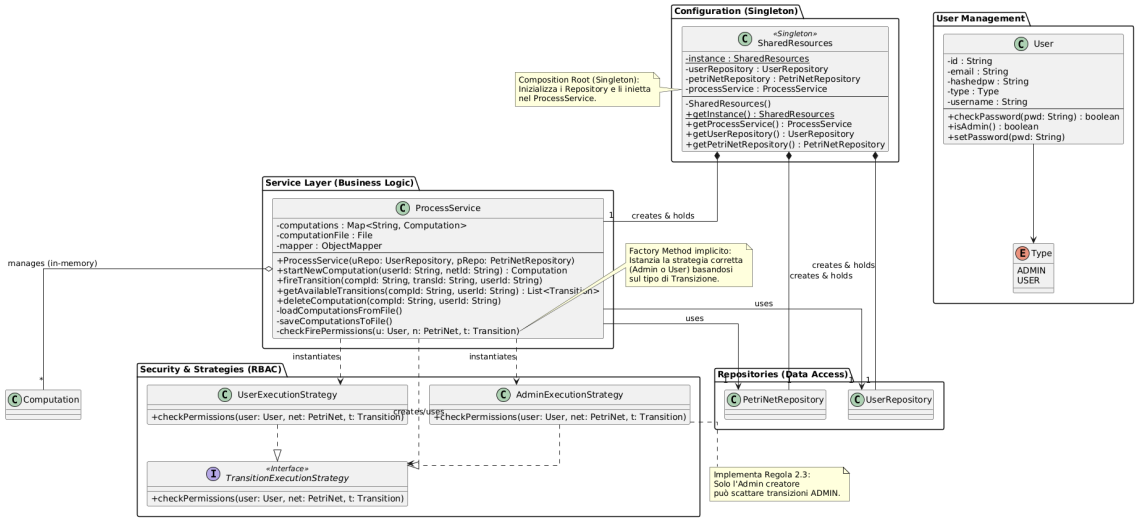

Figure 16: Class Diagram: Architettura Service, Singleton e Strategy.

6 Scelte Implementative

6.1 Persistenza basata su JSON

Per il salvataggio della struttura delle reti e storico delle computazioni, si è optato per una persistenza basata su file locali in formato JSON, scartando soluzioni più pesanti come DBMS relazionali o NoSQL esterni. Questa scelta è dettata dall'esigenza di **semplicità**: l'applicazione risulta immediatamente eseguibile senza richiedere la configurazione di server database.

6.2 Meccanismo di Snapshot

Una delle sfide critiche è la gestione della concorrenza tra l'utente amministratore (che definisce e modifica la struttura della rete) e l'utente finale (che ne esegue una computazione). È stato implementato un meccanismo di **Snapshot tramite Deep Copy**. Invece di collegare la computazione alla rete originale, il sistema clona la `PetriNet` e le relative `PetriNetCoordinates` nel momento esatto in cui la computazione viene istanziata.

I vantaggi principali di questa scelta sono:

- **Isolamento:** L'utente elabora la computazione su una versione "congelata" della rete, e viene quindi garantita l'immutabilità durante l'intera esecuzione, indipendentemente dalle azioni dell'amministratore.
- **Persistenza:** Ogni computazione salvata nel file `computations.json` contiene al suo interno l'intero stato necessario per essere ricostruita, rendendo i dati resistenti alla modifica o all'eliminazione dei template originali.

6.2.1 Serializzazione e Clonazione tramite Jackson

Per realizzare la copia in modo efficiente, si è scelto di sfruttare la libreria **Jackson**. Il `ProcessService` serializza l'oggetto originale in una stringa JSON e lo deserializza immediatamente in una nuova istanza scollegata. Questa tecnica garantisce che tutte le proprietà complesse (Posti, Transizioni, Archi) vengano allocate in nuove e distinte aree di memoria.

6.3 Gestione della Robustezza tramite Eccezioni Custom

Per garantire un'elevata affidabilità del sistema, la validazione si è concentrata nel livello logico. Violazioni come il tentativo di far scattare una transizione senza token sufficienti, o l'accesso non autorizzato da parte di un utente, sollevano **Eccezioni Custom** (es. `TransitionNotEnabledException`, `UnauthorizedAccessException`). I *Controller* intercettano questi errori tramite costrutti `try-catch`, trasformando l'eccezione in un messaggio user-friendly. Questo approccio previene crash improvvisi dell'applicazione e garantisce un feedback costante all'utente.

7 Verifica e Testing

La fase di verifica è stata condotta verso la fine dello sviluppo per verificare la robustezza del sistema e la conformità ai vincoli formali delle Reti di Petri. L'intero processo di build e testing è stato automatizzato tramite **Maven**.

7.1 Strategia di Testing

È stato usato **JUnit 5**. Per isolare la logica di business dalle dipendenze esterne (come il file system), è stata utilizzata la libreria **Mockito**.

- **Unit Testing**: Verifica del comportamento delle classi POJO (*Place, Transition, Arc, User*) e della gestione dello stato (*MarkingData, ComputationStep*).
- **Mocking**: Isolamento del *ProcessService* dai repository tramite oggetti mock, garantendo test indipendenti dai file JSON.

L'automazione tramite Maven ha permesso di verificare ad ogni build che le nuove implementazioni non alterassero la logica preesistente.

7.2 Aree di Copertura dei Test

I test sono stati scritti per coprire tre macro-aree critiche dell'applicazione:

1. **Integrità Strutturale** : Controllo dei vincoli del grafo bipartito (impossibilità di collegare due nodi dello stesso tipo) e verifica della corretta generazione degli identificatori univoci (UUID).
2. **Motore di Esecuzione**: Validazione della regola di scatto all'interno della classe *PetriNet*. È stato testato il corretto consumo e produzione dei token.
3. **Sicurezza e Permessi (Strategy Pattern)**: Verifica delle regole relative ai ruoli. Tramite *UserExecutionStrategy* e *AdminExecutionStrategy*, è stato dimostrato che un amministratore non può aggirare i vincoli operando come utente sulle proprie reti.

7.3 Risultati dell'Esecuzione

L'esecuzione automatizzata non ha riportato alcun fallimento o errore, garantendo la stabilità dell'applicazione. Di seguito è riportato il dettaglio delle classi testate e i risultati della build.

Classe di Test	Test Eseguiti	Falliti	Errori
`AdminExecutionStrategyTest`	3	0	0
`ArcTest`	6	0	0
`ComputationStepTest`	5	0	0
`ComputationTest`	11	0	0
`MarkingDataTest`	10	0	0
`PetriNetTest`	7	0	0
`PlaceTest`	4	0	0
`ProcessServiceTest`	8	0	0
`SharedResourcesTest`	3	0	0
`TransitionTest`	5	0	0
`UserExecutionStrategyTest`	3	0	0
`UserTest`	5	0	0
Totale	**70**	**0**	**0**

Table 1: Riassunto dei test.

L'esito positivo è certificato dal log generato al termine della fase di build:

```
[INFO] Results:
[INFO]
[INFO] Tests run: 70, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.333 s
```