Progetto: **UniClass**	Versione 1.0
Documento: **Change Request**	Data: 17/01/2026

Partecipanti

Nome	Matricola
Cammarota Lucageneroso	NF22500053
Sabetta Giuseppe	NF22500155

REVISION HISTORY	

SOMMARIO

Panoramica del Sistema Attuale

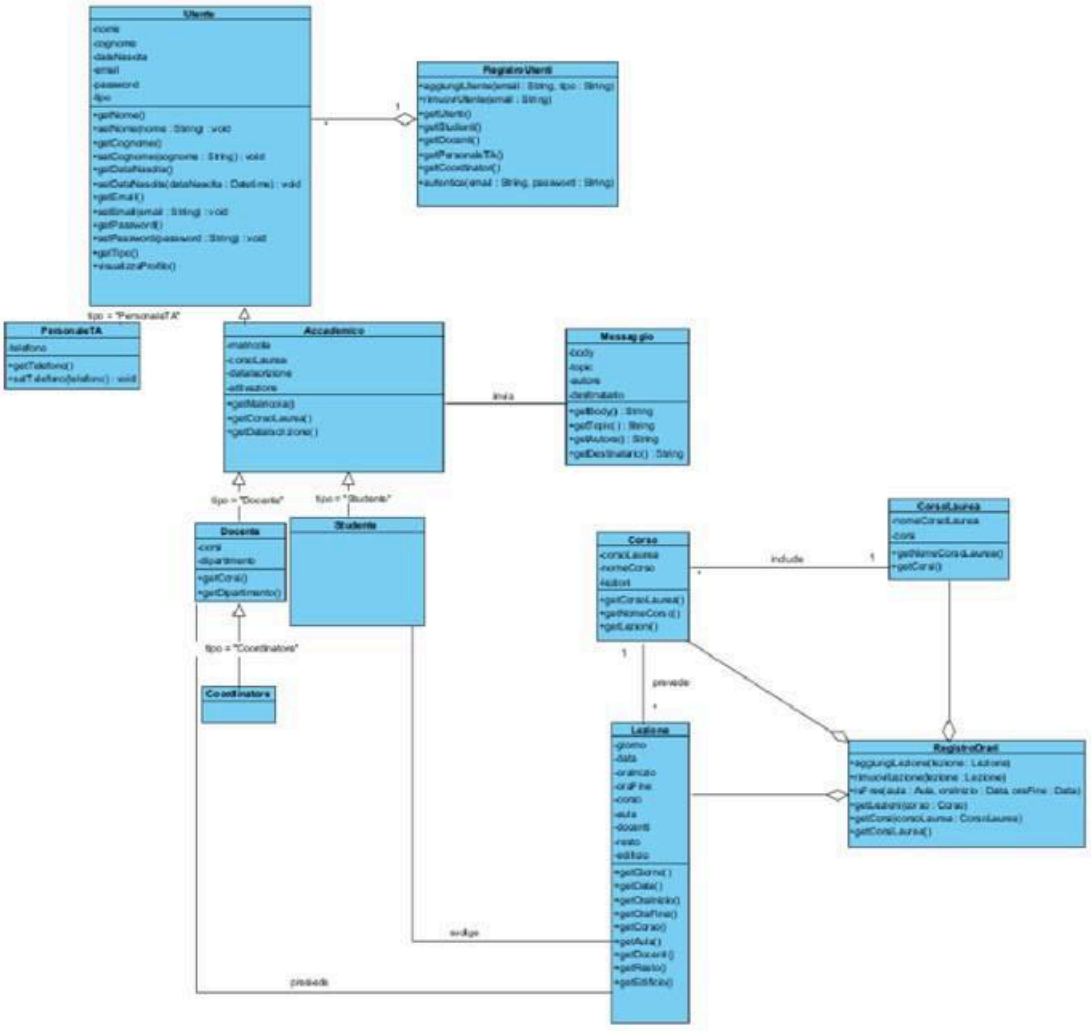
UniClass, Il sistema di gestione universitaria corrente, si basa su un'architettura a **ereditarietà rigida**, progettata per gestire le identità degli utenti e la pianificazione delle attività didattiche attraverso una gerarchia di classi prefissata.

1. Architettura delle Classi (Modello "Is-A")

L'attuale sistema, frutto di un processo di *forward engineering* tradizionale , presenta una gerarchia "Is-A" molto profonda:

- **Modello di Dominio**: Gli utenti sono classificati rigidamente come PersonaleTA o Accademico (a sua volta diviso in Docente e Studente). Questa struttura è "rigida" poiché impedisce a un oggetto di cambiare classe a runtime o di assumere più ruoli contemporaneamente.
- **Persistenza**: Il database utilizza una verticalizzazione dei dati (*Joined Table*), dove i dettagli specifici (es. matricola o dipartimento) sono isolati in tabelle satellite. Ciò richiede costose operazioni di join per ricostruire l'oggetto completo.
- **Registri**: Classi come RegistroUtenti agiscono da "God Object", accentrando logiche di creazione, ricerca e autenticazione che dovrebbero essere distribuite.

L'obiettivo della reingegnerizzazione è migliorare la manutenibilità, attraverso la riduzione del debito tecnico favorendo disaccoppiamento e coesione, e l’estensibilità, permettendo in futuro l’eventuale aggiunta di nuovi ruoli, supportare la gestione multi-ruolo e preparare il sistema a potenziamenti funzionali.



Mapping a oggetti del sistema UniClass attuale

2. Logica di Gestione e Persistenza

Il sistema si appoggia su due componenti centrali per la logica di business:

- **RegistroUtenti:** Si occupa della creazione, rimozione e autenticazione degli utenti. La sua struttura è strettamente legata ai tipi definiti, con metodi specifici per estrarre liste filtrate (es. getStudenti(), getDocenti()).
- **RegistroOrari:** Gestisce la pianificazione delle lezioni e dei corsi, basandosi sul presupposto che una Lezione debba essere obbligatoriamente svolta da un oggetto di tipo Docente.

3. Abstraction

A seguito di una fase di reverse engineering, sono stati analizzati i punti determinanti del technical debt presente nelle gerarchie d'utenza, grazie ai call graph di AccademicoService e AccademicoDAO, più precisamente tutti i metodi, con relative classi, che chiamano i metodi di retrieval di utenti di AccademicoService e AccademicoDAO.

Come visibile dal modello dati presente nel mapping a oggetti del sistema e secondo la knowledge del sistema, nonostante la classe Accademico sia, in realtà, necessariamente astratta, rendendo possibile la sola concretizzazione delle sole sottoclassi rilevanti, è istanziabile ed è **necessaria** per le operazioni CRUD sull'utenza di tipo Accademico, rendendo onerose le query, il codice e flussi di dati relativi.

Dalle immagini in **Allegati e Supporto Tecnico** è possibile visionare la complessità e l'alta coesione dei metodi più importanti delle classi citati poc'anzi con il resto dei servizi codificati nella Web Application UniClass.

Reason for change - Limiti della Struttura Attuale

Nonostante la chiarezza teorica, questo modello presenta criticità operative che impattano sulla scalabilità e sull'aderenza alla realtà accademica:

I call graph **[A.1]** mostrano che **la separazione per ruoli tramite ereditarietà non guida i flussi di esecuzione.** L'analisi evidenzia che i flussi di esecuzione non sono chiaramente separati in base ai sottotipi Docente e Studente, ma convergono sulla classe Accademico, indicando che la distinzione per ereditarietà non guida il comportamento runtime del sistema.

Se in **[A.1]** è presente la parziale evidenza delle dipendenze nelle gerarchie d'utenza, in **[A.4]** è, invece, visibile un importante technical debt nella struttura dei package e del codice: l'adozione del **Service Pattern**, ovvero un *simil-Facade* pattern che implica la gestione della logica computazionale, di business e di controllo in classi diverse da quelle rappresentate dai **DAO Pattern**, prevede un forte accoppiamento tra tutte le classi di business per le operazioni transazionali CRUD – per ogni DAO deve essere disponibile un Service, in grado di garantire una connessione tra il container di TomEE e il servizio PostgreSQL, con ulteriori controlli per ridurre il numero di eccezioni restituite all'utente. Tuttavia, si analizza che non c'è solo un accoppiamento ***intra-service***, dove ogni DAO ha il suo rispettivo Service chiamante, bensì anche ***inter-service***, poichè ogni Service può chiamare più DAO e più Service in un unico metodo e ciò, per quanto possa aumentare la sicurezza (safety) della piattaforma, aumenta il Technical Debt.

Nonostante il Technical Debt descritto nei punti precedenti, con allegato disponibile nella sezione apposita **Allegati e Supporto Tecnico**, sulla base di flow data analysis, query logs, ottimizzazione delle namedQuery di Jakarta seguendo le good practices di **Oracle** e **OWASP**, le operazioni CRUD non rappresentano un onere, nè per la webapp, (data l'ottima gestione di TomEE degli EJB), nè per il database, (grazie all'ORM JPA, configurato per ottimizzare il mapping degli oggetti verso il database e nell'esecuzione delle namedQuery pre-compilate, minimizzando l'overhead a runtime).

Quindi:

- la gerarchia **non aggiunge chiarezza comportamentale**, ma aggiunge complessità strutturale. Ogni cambiamento riguardante l'utenza risulta ostico, longevo e potenzialmente dannoso per l'intera piattaforma. Una modifica nelle operazioni CRUD di qualsiasi sottoclasse di Accademico, (e.g. Docente, Studente, Coordinatore), necessita di un oneroso Impact Analysis per via dell'accoppiamento con la classe Accademico (e viceversa).
- **Analisi del dominio reale. Rigidità dei Ruoli:** Il sistema non permette la gestione di profili "ibridi". Un dottorando che svolge sia attività di studio (Studente) che di insegnamento (Docente) richiederebbe la creazione di due account separati o di una nuova classe specifica, aumentando la ridondanza dei dati.
    - **Outcome ->** Nuovo Requisito Funzionale: il sistema deve supportare la gestione multi-ruolo per un utente.

- **Verticalizzazione della Persistenza:** Attualmente, i dati sono memorizzati in tabelle separate per ogni sottotipo. Questo comporta un alto numero di JOIN nelle query e difficoltà nel mantenere l'integrità referenziale quando un utente cambia status.
- **Documentazione Strutturale Assente**: Per nuovi sviluppatori o revisori del codice sorgente, è difficile determinare la struttura e il flusso dei dati presenti nel contesto dell'utenza.
    - **Outcome**: analisi e riduzione dell'accoppiamento tra le classi previste, come determinato dai punti del reason for change, e redocumentation per migliorare l'explainability.
- **Accoppiamento Forte: Re-Design** I registri (RegistroUtenti, RegistroOrari) sono fortemente accoppiati alle classi concrete. Qualsiasi modifica alla struttura di una classe (es. aggiungere il ruolo "Collaboratore") richiede modifiche invasive in più punti del codice. Queste classi, inoltre, risultano avere troppe responsabilità e metodi.
- **Accoppiamento Services:** i DAO e Service, specialmente dell'utenza, sono fortemente accoppiati e più Service possono chiamare rispettivamente molteplici metodi di DAO e Services di classi presenti in gerarchie tecnicamente non sostenibili.
    - **Outcome**: Durante la ristrutturazione della gerarchia dell'Utenza, è necessario ristrutturare le rispettive classi necessarie per le transazioni CRUD, sia in DAO (collegamento diretto al DB), che service (Service Pattern).

La Change Request che segue propone di superare questi limiti adottando una **struttura a ruoli dinamici** (Composition over Inheritance) e l'Abstract Factory **Pattern** per decentralizzare la gestione degli utenti e favorire la delega.

Description of Change

La modifica proposta consiste in un **re-design dell'architettura del sistema**, accompagnato da una **modifica del modello dati** e da una **parziale reimplementazione e ri-specifica dei requisiti**.

Dal punto di vista architetturale, l'intervento riguarda la **ristrutturazione del sottosistema di gestione degli utenti e dei ruoli**, con particolare attenzione all'entità Accademico.

L'architettura attuale si basa su una gerarchia di ereditarietà rigida (modello *Is-A*), che lega in modo stretto la classificazione strutturale degli utenti alle loro responsabilità comportamentali. Il re-design proposto mira a **disaccoppiare la gestione dei ruoli dalla struttura a classi**, migliorando flessibilità architetturale e manutenibilità nel lungo periodo.

Rethink

L'intervento nasce dalla necessità di rivedere alcune decisioni architetturali adottate nella versione corrente del sistema, che hanno introdotto rigidità strutturale e un incremento del debito tecnico. In particolare, la presenza di componenti con responsabilità multiple (ad esempio gestione e istanziazione degli utenti) e la dipendenza della logica applicativa dal tipo di attore hanno evidenziato limiti in termini di estendibilità, manutenibilità e chiarezza del modello.

In questa fase si è proceduto a un ripensamento del modello concettuale e della distribuzione delle responsabilità, con l'obiettivo di allineare architettura, modello dati e dominio applicativo al comportamento atteso del sistema, evitando soluzioni basate su branching esplicito o logica condizionale dipendente dal ruolo.

Impatto sul modello dati (Rethink → Redesign)

Dal punto di vista del modello dati, l'intervento prevede una revisione della rappresentazione persistente degli utenti accademici e dei relativi ruoli, superando l'attuale schema verticalizzato che introduce vincoli strutturali e rigidità non allineate alle esigenze del dominio applicativo.

In fase di **Rethink**, è stata rivalutata l'assunzione di esclusività dei ruoli a livello di schema, individuandola come una limitazione concettuale che influenza negativamente l'evoluzione del sistema.
A valle di tale analisi, in fase di **Redesign**, il modello dati viene riorganizzato per consentire una rappresentazione più flessibile dei ruoli, riducendo le dipendenze strutturali tra entità e migliorando la capacità del sistema di adattarsi a scenari evolutivi senza impatti invasivi sull'architettura.

Redesign

Il sistema viene sottoposto a una riorganizzazione strutturale dei componenti. In particolare:

Lo scope del sistema viene esteso per supportare una **maggiore variabilità dei ruoli accademici**, consentendo l'introduzione di nuovi ruoli o combinazioni di responsabilità senza richiedere modifiche strutturali invasive. Contestualmente, vengono rimossi vincoli architetturali non necessari imposti dal modello attuale attraverso il collapse delle specifiche della classe "Accademico".

Al fine di non introdurre ulteriore complessità ciclomatica e cognitiva, e con l'obiettivo di ridurne il valore rispetto alla baseline, la variabilità comportamentale viene espressa tramite l'annotazione dei metodi e l'uso della sicurezza dichiarativa del container Jakarta EE (@RolesAllowed), rendendo il ruolo una precondizione di invocazione e non una decisione implementativa.

Inoltre, Il componente RegistroUtenti viene rifattorizzato per dipendere da un'interfaccia astratta (UtenteFactory), introducendo il pattern **Abstract Factory** per redistribuire le responsabilità di creazione e ridurre l'accoppiamento con le classi concrete, favorendo inoltre l'estensibilità.

Re-documentation

Attraverso Traceability Graph, diagrammi UML e descrizioni testuali, si descrive in maniera formale e dettagliata la struttura e il comportamento del sistema, al fine di migliorare la manutenibilità del codice e garantire una chiara comprensione delle relazioni tra i vari componenti. Attraverso una documentazione coerente e sempre aggiornata, facilmente consultabile, si aumenta la *sostenibilità sociale,* per incrementare la produttività degli sviluppatori, e *sostenibilità tecnica*, incrementando la stabilità della piattaforma e riducendo il rischio di introdurre technical debt.

Nel complesso, l'intervento si configura come un'operazione di **manutenzione evolutiva guidata da attività di reingegnerizzazione**, finalizzata ad allineare architettura, modello dati e requisiti al comportamento atteso del sistema e alle esigenze del dominio applicativo, riducendo al contempo la rigidità strutturale e il debito tecnico.

Change Impact

Tasks/Scope Affected (Identification of Candidate Impact Set - CIS)

L'impatto della modifica non è limitato al codice, ma si estende verticalmente su tutti i livelli del ciclo di vita del software, come definito nella metodologia di Impact Analysis:

1. **Requirement Artifacts (RAD):**
    - **UC1 (Autenticazione):** Revisione completa dei flussi alternativi e delle eccezioni a causa del passaggio dalla gerarchia rigida alla gestione dinamica dei ruoli.
    - **UC9 (Creazione Account):** Modifica delle post-condizioni; il sistema non creerà più un'entità "atomica" (Studente o Docente) ma un'entità Utente con ruoli associati.
    - **NFR1 (Sicurezza):** Necessaria ri-validazione dei vincoli di accesso trasversale tra i nuovi ruoli.
2. **Design Artifacts (Object/System Design):**
    - **Object Model:** Sostituzione del modello ereditario "Is-A" con un modello a composizione "Has-A".
    - **Dynamic Modeling:** Aggiornamento dei Sequence Diagram SD1, SD7, SD9 e SD10 per riflettere l'introduzione della UtenteFactory del caricamento dinamico dei profili.
3. **Code Artifacts (Implementazione):**
    - **Core Services: AccademicoServuce** (metodi trovaEmailPassUniclass, retrieveEmail).
    - **Web Layer (Servlet):** LoginServlet, GetEmailServlet, AttivaUtentiServlet (necessitano di adattamento alle nuove interfacce della Factory).
    - **Data Access Layer:** AccademicoDAO, UtenteDAO (ristrutturazione query JPA per eliminare i JOIN onerosi della strategia Joined Table).
4. **Test Artifacts:**
    - Rifattorizzazione integrale di AccademicoServiceTest e LoginServletTest.

Cost Evaluation

Lo sforzo è stimato sulla base del **Remediation Effort** rilevato da SonarCloud. La transizione verso il pattern Abstract Factory e la rimozione della logica "God Object" in RegistroUtenti richiederà un impegno significativo per garantire che la **Precision** dell'analisi sia alta (evitando modifiche non necessarie a moduli stabili come RegistroOrari, se non per la firma dei parametri).

Risk Evaluation (Ripple Effect)

Il rischio è classificato come **ALTO** per i seguenti fattori metrici:

- **Cyclomatic Complexity (575):** L'elevato numero di percorsi indipendenti rende difficile garantire una **Recall** del 100% (rischio di False Negatives: bug che sfuggono all'analisi iniziale).
- **Cognitive Complexity (219):** L'attuale logica annidata aumenta la probabilità di "Side Effects" durante lo sradicamento della gerarchia Accademico.
- **Change Propagation:** Una modifica ai metodi di retrieval in AccademicoDAO ha un effetto "ripple" che risale fino alle JSP della UI.

Quality Evaluation

L'intervento porta a un miglioramento tangibile della qualità del sistema misurabile tramite indicatori oggettivi, garantendo maggiore manutenibilità, minore complessità cognitiva, riduzione del debito tecnico e miglior testabilità, senza compromettere la stabilità e le performance attuali.

Le metriche di qualità a cui si fa riferimento sono riportate in dettaglio nella sezione **A.3 Indicatori di Complessità (Baseline SonarCloud)** e nella sezione **A.4 Indicatori di Tech Debt tramite Analisi Automatizzate**.

Queste metriche sono utilizzate per valutare il debito tecnico accumulato, indicando il rischio e l'effort necessario per la manutenzione futura, e costituire una baseline di confronto per analizzare l'efficacia dell'intervento manutentivo.

Duration

La durata del completamento, (e documentazione del razionale a-priori), delle task designate in **Description of Change** è stimata per il 12 Febbraio 2026. Per la gestione delle risorse temporali da allocare agli strumenti predisposti per il refactoring si utilizza **Trello**, piattaforma KanBan Dev-Ops, come strumento di supporto alla suddivisione delle attività e le **milestones** da raggiungere, attraverso Diagrammi Gantt presenti in piattaforma.

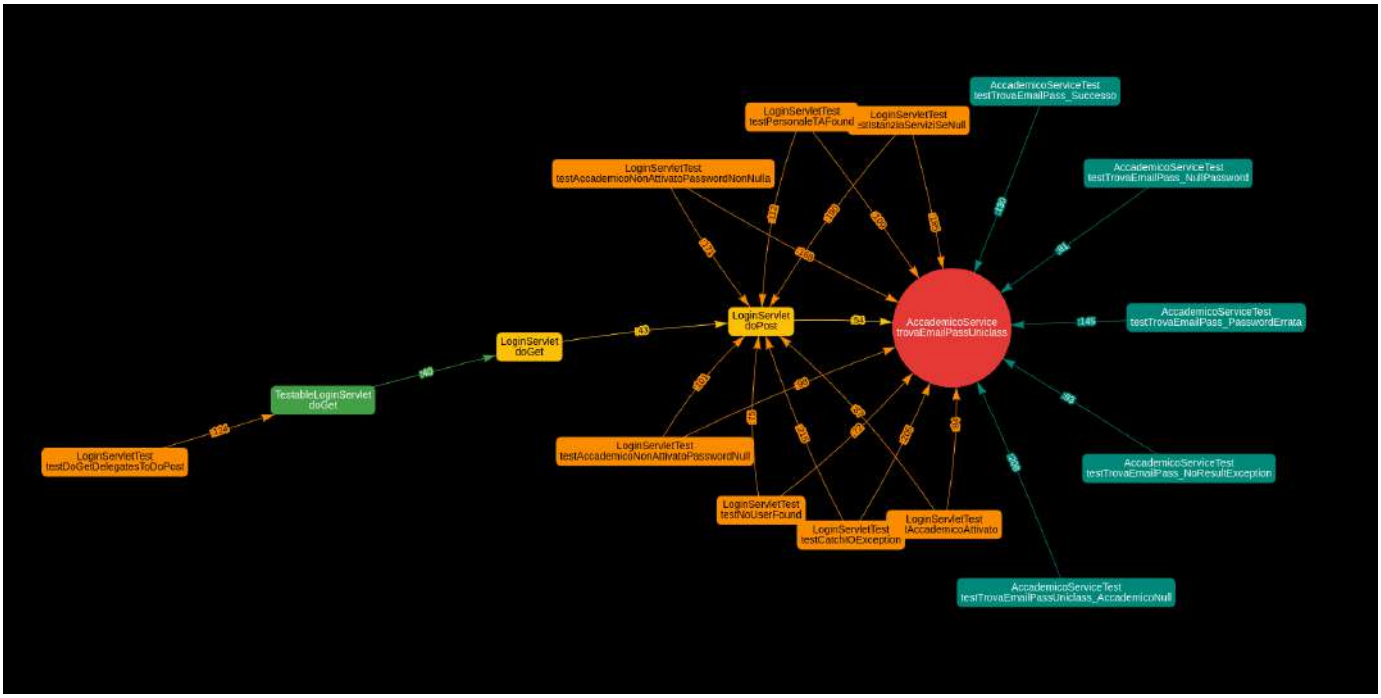
Allegati e Supporto Tecnico

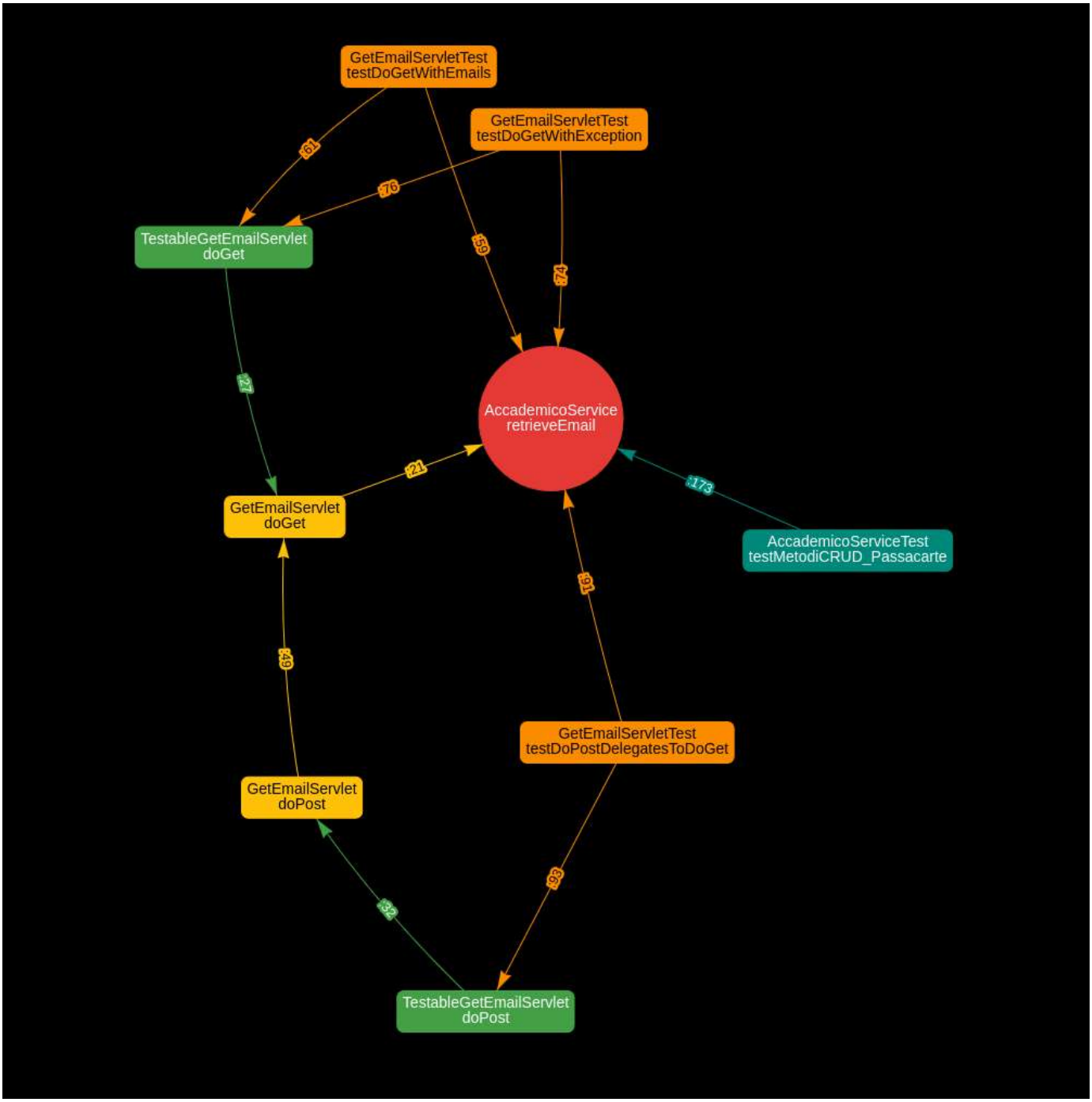
In questa sezione vengono riportati gli artefatti grafici fondamentali per la comprensione del debito tecnico attuale e della propagazione del cambiamento proposto.

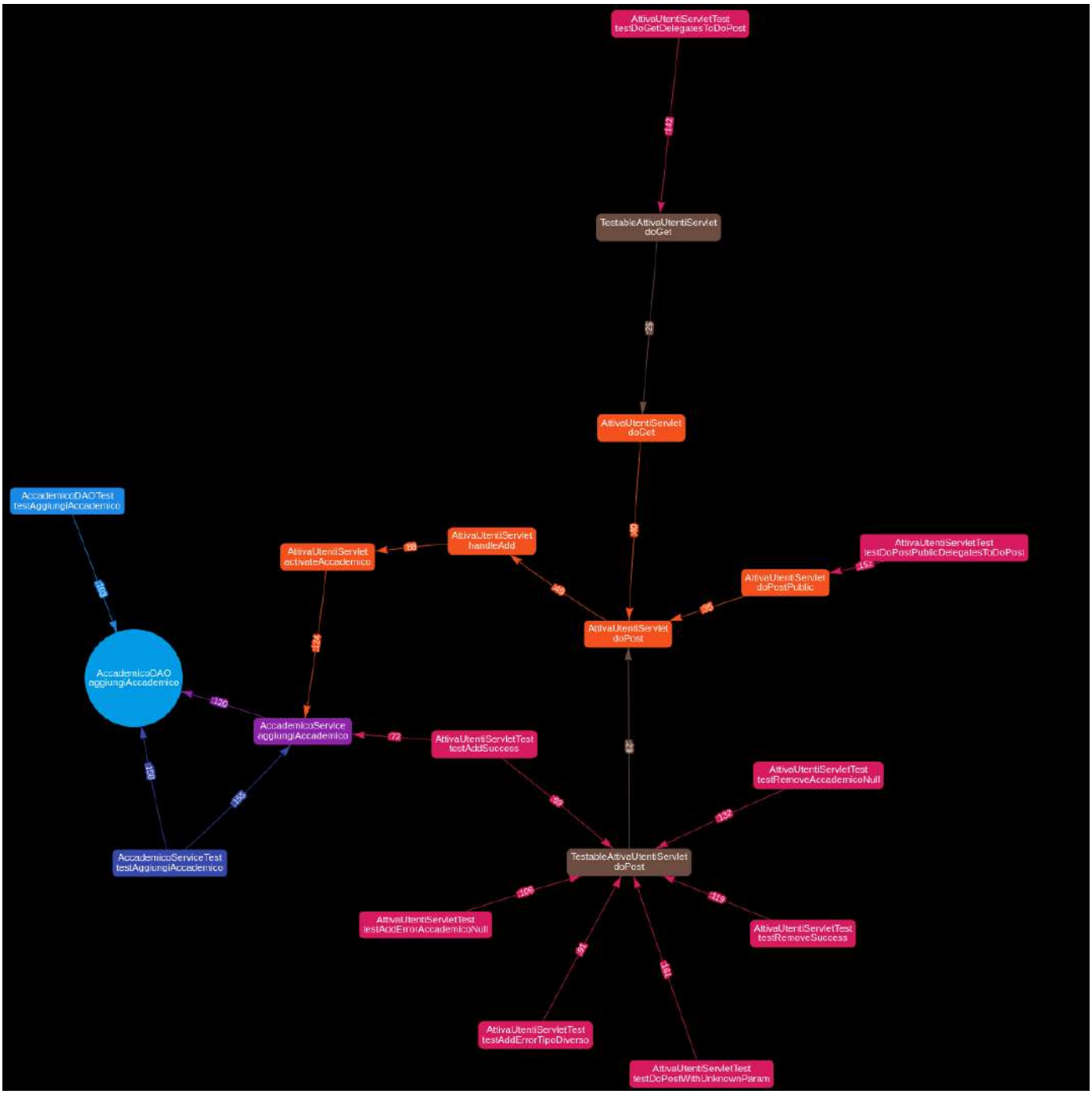
A.1 Call Graph dei Metodi Hotspot

I seguenti grafi rappresentano le interazioni dinamiche rilevate in fase di reverse engineering. Come evidenziato nell'analisi, si nota un'altissima densità di chiamate che convergono verso i metodi di retrieval di AccademicoService, indipendentemente dal ruolo logico dell'utente.

- **Hotspot Login:** Mappa delle chiamate verso trovaEmailPassUniclass.
- **Hotspot Recupero Dati:** Mappa delle chiamate verso retrieveEmail.
- **Hotspot Creazionale**: Mappa delle chiamate da/verso aggiungiAccademico

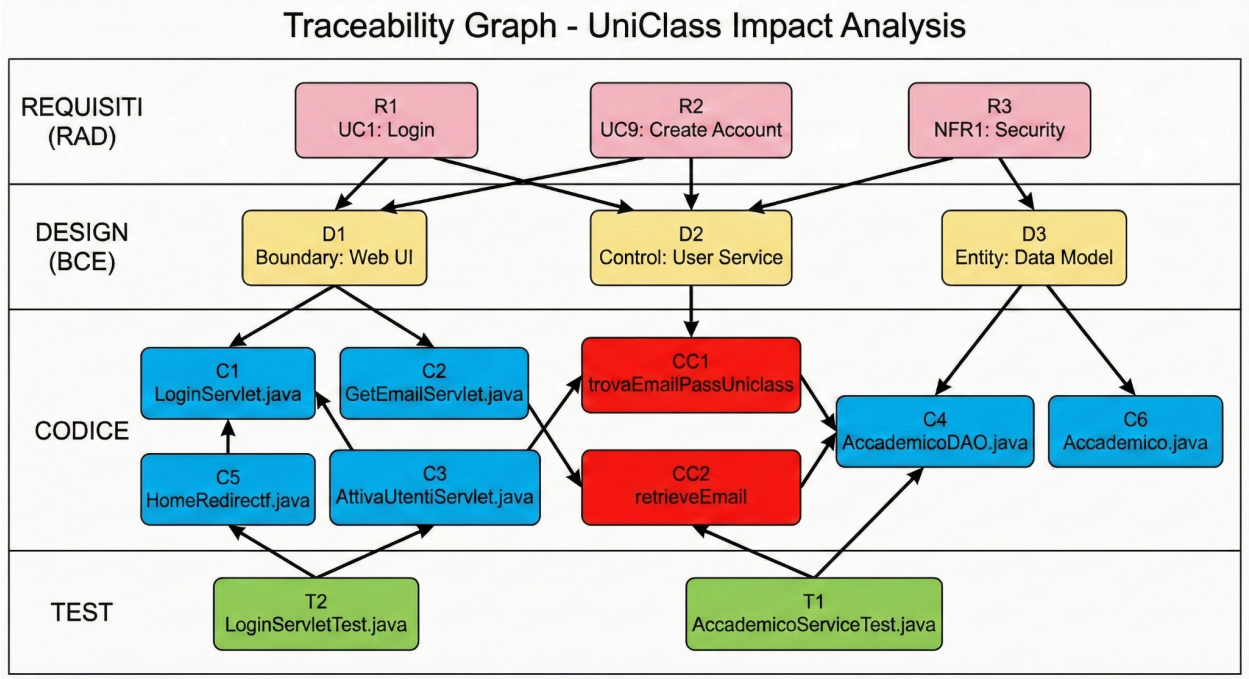

AccademicoServiceTest
testTrovaEmailPass_Successo
LoginServletTest
testPersonaleTAFound
LoginServletTest
istanziaServiziSeNull
LoginServletTest
testAccademicoNonAttivatoPasswordNonNulla
AccademicoServiceTest
testTrovaEmailPass_NullPassword
LoginServlet
doPost
AccademicoService
trovaEmailPassUniclass
AccademicoServiceTest
testTrovaEmailPass_PasswordErrata
LoginServlet
doGet
TestableLoginServlet
doGet
LoginServletTest
testDoGetDelegatesToDoPost
LoginServletTest
testAccademicoNonAttivatoPasswordNull
AccademicoServiceTest
testTrovaEmailPass_NoResultException
LoginServletTest
testNoUserFound
LoginServletTest
testCachiOException
LoginServletTest
tAccademicoAttivato
AccademicoServiceTest
testTrovaEmailPassUniclass_AccademicoNull


GetEmailServletTest
testDoGetWithEmails
GetEmailServletTest
testDoGetWithException
61
76
56
74
TestableGetEmailServlet
doGet
72
AccademicoService
retrieveEmail
24
173
GetEmailServlet
doGet
AccademicoServiceTest
testMetodiCRUD_Passacarte
91
69
GetEmailServletTest
testDoPostDelegatesToDoGet
GetEmailServlet
doPost
93
32
TestableGetEmailServlet
doPost


AttivaUtentiServletTest
testDoGetDelegatesToDoPost
TestableAttivaUtentiServlet
doGet
AttivaUtentiServlet
doGet
AccademicoDAOTest
testAggiungiAccademico
AttivaUtentiServlet
handleAdd
AttivaUtentiServlet
activateAccademico
AttivaUtentiServletTest
testDoPostPublicDelegatesToDoPost
AttivaUtentiServlet
doPostPublic
AttivaUtentiServlet
doPost
AccademicoDAO
aggiungiAccademico
AccademicoService
aggiungiAccademico
AttivaUtentiServletTest
testAddSuccess
AttivaUtentiServletTest
testRemoveAccademicoNull
AccademicoServiceTest
testAggiungiAccademico
TestableAttivaUtentiServlet
doPost
AttivaUtentiServletTest
testAddErrorAccademicoNull
AttivaUtentiServletTest
testRemoveSuccess
AttivaUtentiServletTest
testAddErrorTipoDiverso
AttivaUtentiServletTest
testDoPostWithUnknownParam

A.2 Traceability Graph Multi-Layer (Impatto della Modifica)

Il grafico sottostante illustra la tracciabilità del cambiamento attraverso i diversi layer del ciclo di vita del software.



Legenda del Grafico:

- **REQUISITI (Rosa):** Casi d'uso e vincoli di sicurezza dal RAD impattati.
- **DESIGN (Giallo):** Sequence Diagram (SD1, SD7, SD9, SD10) e l'**Object Model** che verrà ristrutturato per supportare la gestione multi-ruolo.
- **IMPLEMENTAZIONE (Blu/Rosso):** I nodi in **rosso (CC)** indicano i candidati primari alla modifica (Change Candidates).
- **TEST (Verde):** Gli artefatti di verifica che garantiscono la *Recall* dell'analisi di impatto.

A.3 Indicatori di Complessità (Baseline SonarCloud)

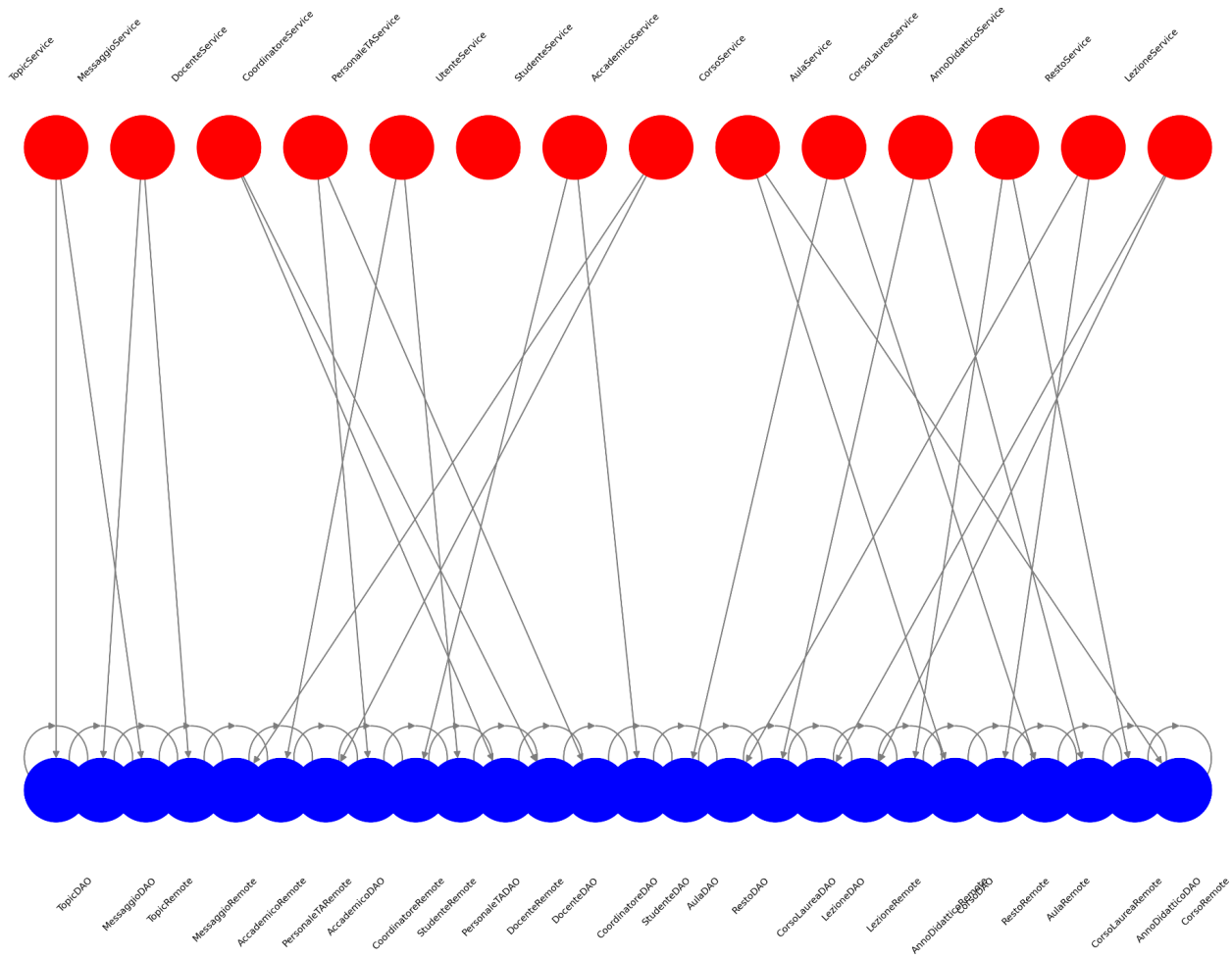
Tabella riassuntiva dei valori metrici prima dell'intervento di reingegnerizzazione:

Metrica	**Valore Baseline**	**Soglia di Allerta**
Cyclomatic Complexity	**575**	> 100 (modulo)
Cognitive Complexity	**219**	> 15 (metodo)
Technical Debt Ratio	**Rating A**	< 5% (Rating A)

A.4 Indicatori di Tech Debt tramite Analisi Automatizzate

Per motivi didattici e ingegneristici, si è deciso di incrementare l'analisi del **debito tecnico**, al fine di incrementare la sostenibilità tecnica, sociale e ambientale della piattaforma, attraverso diverse metriche e tool.

Come prima metodologia è uno script python creato manualmente per visionare l'accoppiamento **intra** e **inter** service/dao.



Come **seconda metodologia**, aggiornando i file di configurazione del container di TomEE, è possibile determinare il tempo richiesto da ciascuna operazione del database. Nonostante il technical debt presente a livello di design, l'analisi delle query log mostra che la maggior parte delle query viene eseguita in tempi ottimali, specialmente nelle query core. Le uniche eccezioni sono presenti nello start-up del progetto, dove alcune query di inizializzazione richiedono più tempo del previsto (la prima query richiede 18ms, mentre la create dei messaggi è 20ms, con i successivi statements che si attestano intorno ai 2-3ms). Questo suggerisce che, nonostante il debito tecnico a livello di progettazione, le performance del database sono comunque accettabili per la maggior parte delle operazioni quotidiane. L'ottimalità è garantita (in parte) anche grazie alla correttezza della codifica delle query parametriche in Java, le entità ben modellate e l'ottimizzazione delle relazioni tra le tabelle del database e le chiamate del service layer.