Università degli Studi di Salerno

Corso di Ingegneria del Software Tecniche Avanzate

UniClass

Test Plan

Gruppo di Lavoro:

Cognome e Nome	**Matricola**
Cammarota Lucageneroso	NF22500053
Sabetta Giuseppe	NF22500155

22 febbraio 2026

Indice

Capitolo 1

Introduction

Il presente documento descrive il Piano di Test (Test Plan) per l'intervento di manutenzione evolutiva e correttiva sul sistema software **UniClass**. L'intervento, definito nel documento *Change Request (CR-UniClass)*, prevede un refactoring architetturale del modello di gestione utenti e un'ottimizzazione delle performance di accesso al database in fase di start-up.

1.1 Scopo del documento

L'obiettivo di questo piano è definire la strategia di verifica atta a garantire che:

- Il passaggio dal modello a ereditarietà (Is-A) al modello a composizione/strategy (Has-A) non introduca regressioni funzionali.
- Le query di inizializzazione del database e di creazione messaggi rispettino i nuovi vincoli prestazionali.
- La qualità del codice (Coverage, Mutation Score) rimanga in linea con gli standard elevati raggiunti nella versione precedente (Baseline).

Capitolo 2

References

I seguenti documenti sono stati utilizzati come riferimento per la stesura del piano:

- **CR-UniClass (v1.0):** Documento di richiesta di modifica e analisi di impatto.
- **UniClass - Report SwD (Baseline):** Report che certifica lo stato di dependability della versione attuale del software.
- **IEEE Std 829-2008:** Standard for Software and System Test Documentation.
- **Repository GitHub:** `uniclass-dependability`.

Capitolo 3

Test Items

In accordo con la Change Impact Analysis, i seguenti artefatti software sono oggetto di test:

3.1 Componenti Oggetto di Refactoring (High Impact)

Questi componenti subiranno modifiche strutturali radicali e richiedono test "White Box" approfonditi.

- **Package Model:** `it.unisa.uniclass.utenti.model.*`
  - Classi target: `Utente`, `Studente`, `Docente`, `PersonaleTA`.
  - Obiettivo: Verifica della nuova gestione dei Ruoli tramite pattern Strategy/Composite.
- **Package DAO:** `it.unisa.uniclass.utenti.service.dao.*`
  - Obiettivo: Verifica del mapping ORM (JPA) sulle tabelle dopo la rimozione dell'ereditarietà `JOINED` o `SINGLE_TABLE`.

3.2 Componenti Oggetto di Regressione (Medium/Low Impact)

Componenti che utilizzano i moduli rifattorizzati e devono essere testati in modalità "Black Box" o integrazione.

- **Service Layer:** `UtenteService`, `MessaggioService`.
- **Controller:** `LoginServlet`, `AttivaUtentiServlet`.
- **Configurazione:** `DatabasePopulator` (Oggetto di ottimizzazione performance).

Capitolo 4

Software Risk

Sulla base dello stato attuale del progetto (Baseline), sono stati identificati i seguenti rischi critici per la fase di test:

1. **Obsolescenza della Test Suite (Rischio Alto):** La suite di test attuale istanzia direttamente classi concrete (es. `new Studente()`). Con il refactoring, queste classi potrebbero cessare di esistere o cambiare semantica, causando errori di compilazione massivi nella suite esistente.
2. **Regressioni nel Mapping ORM:** Il cambio di strategia di ereditarietà in JPA (da ereditarietà a composizione) è complesso. Esiste il rischio concreto di perdita di dati o errori nelle query JPQL esistenti se non aggiornate correttamente.
3. **Race Conditions (Docker):** Come evidenziato nel Report SwD precedente, l'ambiente containerizzato ha mostrato in passato problemi di race condition tra PostgreSQL e l'applicazione. L'ottimizzazione dello start-up time potrebbe riacutizzare questo problema.

Capitolo 5

Features to be tested

5.1 Funzionalità

- **Autenticazione (Login/Logout):** Verifica corretto accesso per tutti i ruoli dopo il cambio strutturale.
- **Gestione Utenti:** CRUD utenti (creazione, lettura, aggiornamento stato attivazione).
- **Messaggistica:** Invio e ricezione messaggi (verifica impatto indiretto tramite chiave esterna utente).

5.2 Validazione dell'Impatto Architetturale sulle Performance

Il passaggio da un'architettura basata su ereditarietà (*Is-A*) a una basata su composizione (*Has-A*) modifica sostanzialmente la struttura delle query generate dall'ORM (JPA). I test prestazionali mirano a validare che questa semplificazione strutturale si traduca in un effettivo vantaggio computazionale.

- **Ottimizzazione Query di Startup (Reduction of Join Complexity):** La nuova architettura deve eliminare l'overhead causato dalle strategie di mapping dell'ereditarietà (es. `JOINED` o `SINGLE_TABLE` sparse).
  - *Oggetto:* Query di inizializzazione DB e popolamento iniziale.
  - *Target:* Tempo di esecuzione $< 15ms$ (riduzione concreta rispetto alla baseline di 18-20ms dovuta alla semplificazione dello schema relazionale).
- **Efficienza del Retrieve (Throughput Operazioni):** Verifica che il recupero delle liste utenti tramite il nuovo pattern *Strategy/Composite* non introduca latenze dovute alla navigazione delle associazioni.
  - *Oggetto:* Metodi `findAll` e filtri per ruolo in `UtenteService`.
  - *Target:* Throughput invariato o superiore ($Ops_{Post} \geq Ops_{Baseline}$), confermando che l'assemblaggio degli oggetti in memoria è più rapido o equivalente rispetto alla materializzazione delle sottoclassi.

Capitolo 6

Features not to be tested

Le seguenti aree non saranno oggetto di test specifici in questa iterazione, a meno di evidenti regressioni bloccanti:

- **Interfaccia Utente (UI/CSS):** Il refactoring è esclusivamente backend; non sono previste modifiche al frontend JSP/CSS.
- **Modulo "Edifici e Aule":** Basso accoppiamento con la gerarchia utenti; verranno eseguiti solo smoke test.
- **Librerie di terze parti:** (es. Driver JDBC, librerie Maven) non verranno testate isolatamente.

Capitolo 7

Test Strategy and Approach

La strategia di testing seguirà un approccio *Test-Driven Maintenance*, adattando la suite esistente alle modifiche architetturali. Il piano si articola in due macro-aree: **Functional Testing**, verificando la correttezza funzionale post-refactoring, e **Non-Functional Performance Testing**, per la validazione dei benefici prestazionali attesi dalla semplificazione architetturali schedulati per la manutenzione correttiva e perfettiva del sistema.

7.1 Overview

L'intervento di manutenzione comporta un cambio radicale del modello dati (da Is-A a Has-A). Per garantire la qualità del sistema successivo all'intervento di manutenzione preposto, si adotterà la strategia di testing seguente:

- **Regression Testing Continuo**: Durante il refactoring, il codice viene riorganizzato, semplificato e ottimizzato e, per questo motivo, potrebbero esserci modifiche pronte ad introdurre involontariamente dei bug. Di conseguenza, ogni volta che si modifica il codice (modifiche leggere ed incrementali), la suite di test deve essere eseguita automaticamente. La suite di test attuale funge così da *Safety Net*.
- **Test Repair**: Si adatterà progressivamente la suite di test unitari per riflettere in maniera iterativa e incrementale, conseguente a quanto detto in precedenza, la nuova struttura del codice, migliorata e refactorizzata.
- **Performance Validation**: Si effettuerà un confronto sistematico tra la baseline pre-intervento e metriche post-refactoring tramite microbenchmark, per i test d'unità, e benchmark di sistema, attraverso JMeter, per il raggiungimento del goal degli attributi di Software Dependability.

7.2 Functional Testing

7.2.1 Scope and Objectives

Il testing funzionale ha quindi il compito di garantire che la rimozione della gerarchia di ereditarietà e l'introduzione dei pattern non introducano regressioni nelle funzionalità

core (e non) del sistema. In particolare si gestiranno, dal punto di vista implementativo e, di conseguenza, di test funzionale, i seguenti aspetti:

- Autenticazione
- Operazioni CRUD sugli utenti (principalmente l'attivazione di quest'ultimi)
- Gestione dei ruoli accademici e di segreteria
- Eventuale impatto della messaggistica (indiretto per FK Utente)

7.2.2 Test Levels

Unit Testing

L'approccio sarà white-box testing su classi individuali della totalità dei package presenti nel codice sorgente Backend della piattaforma UniClass. I mock statici verranno sostituiti o aggiornati per riflettere la composizione degli oggetti post-intervento. Si cita così la **Configurazione del Testing**:

- **Framework**: JUnit Jupiter (JUnit 5) - JUnit 5.11.4; Mockito 5.11.0;
- **Build Tool**: Maven 3.9.11 (Red Hat 3.9.11-5);
- **JDK**: Eclipse-Temurin: JDK 21 (Produzione) OpenJDK 21.0.9 (Testing).

System Testing

Si effettua un approccio Black-box end-to-end tramite interfacce web designate attraverso JSP. e con un **ambiente configurato** nel modo seguente:

- **Tool**: Selenium IDE 6.22.0
- **Browser**: Firefox 145 per Nobara - Fedora
- **Target**: TomEE 10 Plume in container Docker

Poichè il refactoring è interno al backend e non modifica l'interfaccia utente, i test selenium possono essere eseguiti senza alcuna modifica. Verranno effettuati i test sull'autenticazione e sull'attivazione degli utenti, rispetto allo Unit Test che testerà tutte le unità presenti all'interno della totalità dei package di sviluppo.

7.3 Strategia di Regression Testing

Considerata la natura invasiva del refactoring (modifica della gerarchia delle classi core `Utente`), l'approccio di regressione non potrà limitarsi al semplice "Retest-All", poiché la compilazione stessa dei test unitari originali fallirà. Verrà adottata una strategia ibrida definita **Test-Repair Regression Strategy**, strutturata come segue:

7.3.1 Regressione a Livello di Sistema (Black Box Safety Net)

Poiché il refactoring modifica il backend ma non l'interfaccia utente (JSP/Servlet interface), i Test di Sistema esistenti fungeranno da "rete di sicurezza" primaria.

- **Approccio:** *Retest-All*.
- **Tool:** Selenium IDE / Katalon Recorder (eseguendo gli script `.bat` presenti in `Report/`).
- **Obiettivo:** Verificare che, nonostante il cambio del modello dati, le funzionalità E2E (es. "Uno studente effettua il login") producano lo stesso risultato visibile all'utente.
- **Criterio di Successo:** I test devono passare senza modifiche agli script di test.

7.3.2 Regressione a Livello Unitario (White Box Adaptation)

A livello di codice sorgente, l'ereditarietà (es. `Studente extends Utente`) verrà rimossa. Questo rende obsoleti i test unitari esistenti che sfruttano il polimorfismo.

- **Approccio:** *Selective Regression with Test Refactoring*.
- **Azione:** I test non verranno eseguiti ciecamente. Verranno identificati i test case impattati (tramite *Impact Analysis*) e adattati al nuovo pattern (es. sostituire `instanceof` con `getRuolo()`).
- **Ambito (Scope):**
  - `it.unisa.uniclass.testing.unit.utenti.*` (Impatto Critico)
  - `it.unisa.uniclass.testing.integration.*` (Impatto Alto su DAO)

7.3.3 Prioritizzazione dei Test (Test Tiers)

Per ottimizzare i tempi di esecuzione durante il ciclo di sviluppo CI/CD (GitHub Actions), la suite di regressione è stata suddivisa in livelli di priorità:

Livello (Tier)	Descrizione e Scope	Esecuzione
Tier 1: Smoke	**Critical Path:** Startup del Database, Login Servlet, Connessione DB. Verifica l'assenza di *Configuration Drifts*.	Ad ogni Commit (CI)
Tier 2: Target	**Refactored Features:** CRUD Utenti, Gestione Ruoli (ex-sottoclassi), DAO layer con nuove query JPQL.	Ad ogni Commit su branch di feature
Tier 3: Full	**Full Regression:** Include test su messaggistica, aule e gestione corsi (impattati indirettamente dalle chiavi esterne).	Nightly Build / Pre-Release

Tabella 7.1: Strategia di Prioritizzazione della Regressione

7.3.4 Automazione e Continuous Integration

Il processo di regressione sarà integrato nella pipeline esistente di GitHub Actions (`maven-ci.yml`).

1. **Build Check:** Verifica che il nuovo modello compili correttamente.
2. **Unit Tests (Surefire):** Esecuzione Tier 1 e Tier 2. Il fallimento blocca la PR.
3. **Performance Check (JMH):** Esecuzione automatica dei benchmark di startup per verificare la conformità con i requisiti non funzionali ($< 15ms$).

7.4 Non Funzionale

7.4.1 Scope and Objectives

Il performance testing ha l'obiettivo di validare sperimentalmente che la risoluzione architetturale e implementativa delineata dall'intervento di manutenzione correttiva e perfettiva siano un vantaggio importante prestazionale misurabile e, in caso contrario, esporre le motivazioni e trade-offs.
Gli **obiettivi specifici** sono i seguenti:

- **Ottimizzazione Query**: Ridurre il tempo sia dell'inizializzazione del database attraverso le modifiche presenti nel modello dati con generazione di strategie di mapping JPA, sia delle query più semplice presentate dalla modifica del modello dati dal punto di vista architetturale
- **Throughput Invariante**: Assicurare che il throughput delle operazioni CRUD sull'utenza, dal punto di vista d'autenticazione e d'attivazione, non degradi a causa della nuova struttura di navigazione delle query e delle join nei database (`Utente -> Ruolo`). Inoltre, indipendentemente da peggioramenti o miglioramenti dettati dalla modifica architetturale, evidenziare trade-offs e Rationale.
- **Resilienza, Disponibilità, Affidabilità**: Verificare che gli attributi chiave dei goal di Software Dependability principali siano effettivamente presenti e soddisfatti in diversi scenari presentati dal tool, che si analizzerà in successione, JMeter, con scenari di stress-test, spike-test in diversi ambienti principali, con Rationale.

7.4.2 Performance Engineering Pipeline

La validazioen prestazionale seguirà un processo sistematico articolato in quattro fasi principali:

1. **Baseline Measurement**: Misurazione delle performance del sistema prima del refactoring presentato dall'intervento di manutenzione.
2. **Refactoring Implementation**: Applicazione delle modifiche architetturali attraverso adozione dei diversi pattern descritti in precedenza.
3. **Post-Refactoring Benchmark**: Riesecuzione degli stessi test di performance sulla nuova architettura.

4. **Comparative Analysis**: Confronto statistico delle metriche pre e post.

Per la metodologia di misurazione, si sono utilizzati questi tool:

- MicroBenchmark - Attraverso JMH (Java Microbenchmark Harness) per misurazioni isolate sulla totalità dei services che compongono il sistema lato Backend. Ogni benchmark viene eseguito con warmup (pre-riscaldamento JVM) e multipli fork per eliminare bias da JIT compilation e garbage collection;
- **Spike & Stress** - Attraverso l'utilizzo di JMeter, è possibile creare scenari Enterprise realistici con carico concorrente basati sulle funzionalità core presenti nella piattaforma.

7.4.3 Environment Setup

Per garante riproducibilità dei risultati, i benchmark JMH e JMeter verranno eseguiti in un ambiente standardizzato e sterile, privo di ulteriori servizi per consentire una misurazione efficace (con la consapevolezza dei limiti dei software-based tools per misurazione).

- **OS**: Nobara Linux 43 (KDE Plasma Desktop Edition) $x86_64$
- **Kernel**: Linux 6.18.7-200.nobara.fc43.$x86_64$
- **CPU**: AMD Ryzen 5 3500 - 6 core @4.12 GHz
- **GPU**: AMD Radeon RX 5500XT
- **Memory (Physical)**: 16 GB DDR4
- **Memory (Swap-Digital)**: 8 GB

7.4.4 Baseline Definition

La baseline presenta diverse metriche estratte dall'esecuzione del sistema nella sua configurazione attuale (architettura con eccessiva gerarchia ed ereditarietà complessa), certificata nella directory, presente nella repository, come un insieme di json (per i microbenchmark) e csv (per i test di sistema) descritti in seguito.

7.4.5 Microbenchmark Testing

Rationale

JMH (Java Microbenchmark Harness) è stato scelto come tool primario per il performance testing a livello di metodo per via della consapevolezza intrinseca degli effetti di basso livello come JIT compilation, branch prediction, CPU cache warming e permette di misurare singoli metodi, eliminando rumore da componenti non coinvolte. Esso è utile per la riproducibilità, importante per la comprensione delle differenze statistiche a livello pre-post refactoring dato dall'intervento di manutenzione, garantendo risultati robusti. Esso verrà utilizzato per comprendere l'efficienza energetica nell'esecuzione dei metodi presenti nel Service Layer Pattern, di conseguenza l'esecuzione di tutti i metodi delle classi Service, in un ambiente controllato.

JMH Configuration

La seguente configurazione sarà utilizzata (con eventuali intorni in base alla motivazione dietro alla misurazione dei metodi specificati) come segue:

Parametro	Valore / Descrizione
JMH Version	1.37 (tramite dependency Maven)
Mode	`ALL` (Tutte le modalità operazionali presenti)
Warmup Iterations	5-10 iterazioni × 1 secondo (per stabilizzare JIT compiler)
Measurement Iterations	5-10 iterazioni × 1 secondo (raccolta dati effettivi)
Forks	2-3 (esecuzione in 2-3 JVM separate per eliminare bias da run precedenti)
Output Format	JSON
Thread Count	1 (single-thread, per isolare effetti di concorrenza)

Tabella 7.2: Configurazione JMH per Microbenchmark

Risultati Baseline

I risultati completi dei benchmark JMH (formato JSON) sono disponibili nella directory presente nella root della repository. Di seguito si riportano le note sulle colonne (ed eventuale spiegazione dei valori) e tabelle riassuntive e dettagliate per le categorie più rilevanti rispetto all'intervento di refactoring.
Note:

- **Throughput (ops/us)**: Operazioni per microsecondo dove i valori più alti indicano performance migliori;
- **Avg Time (us/op)**: Tempo medio di esecuzione per operazione in microsecondi;
- **Sample (us/op)**: Tempo campionato per operazione con l'inclusione della variabilità per l'ambiente a runtime;

Il sommario di esecuzione di JMH è il seguente:

Categoria	Throughput (ops/us)	Avg Time (us/op)	Sample Time (us/op)
Anno Didattico	0.223	4.503	5.539
Aula	0.223	4.534	5.838
Coordinatore	821.643	0.002	0.027
Corso Laurea	0.228	4.429	5.569
Docente	692.139	0.002	0.028
Lezione	0.217	4.635	5.371
Messaggio	0.221	4.522	6.546
Personale TA	494.982	0.002	0.027
Resto	0.220	4.545	5.488
Topic	0.220	4.566	6.165
Utente	181.690	0.006	0.031

Tabella 7.3: Sintesi Risultati Benchmark JMH per Categoria (Baseline Pre-Refactoring)

Dalla tabella è possibile visionare le categorie che mostrano un throughput significativamente superiori rispetto alle altre categorie, indicando operazioni ottimizzate a livello di query e logica applicativa. Nonostante le problematiche riscontrate nella comprensione dell'architettura gerarchia nella change request, le categorie con throughput più basso sono Topic, Messaggio, Utente e Lezione, per via delle operazioni con query con maggiore overhead nel database. Una **nota particolare** è che l'architettura gerarchica dà, per definizione, maggiori problemi quando ci si interfaccerà con i test di sistema presenti con JMeter.

Le tabelle seguenti riportano risultati ben dettagliati per le categorie più critiche per JMH (e per refactoring architetturale). Ogni valore è accompagnato dall'errore statistico e dall'ambiente di runtime. I calcoli sono su 3 fork con 10 iterazioni di misurazione.

Benchmark	**Throughput (ops/us)**	**Avg Time (us/op)**	**Sample Time (us/op)**
`LoginAccademico`	141.27 ± 0.19	0.00709 ± 0.00001	0.032 ± 0.002
`LoginFallito`	139.31 ± 0.15	0.00720 ± 0.00003	0.032 ± 0.002
`LoginPersonaleTA`	264.49 ± 0.57	0.00378 ± 0.00001	0.028 ± 0.001

Tabella 7.4: Dettaglio Benchmark JMH - Categoria Utente (Baseline)

Benchmark	**Throughput (ops/us)**	**Avg Time (us/op)**	**Sample Time (us/op)**
`TrovaPerCorso`	298.17 ± 3.04	0.003 ± 0.000	0.029 ± 0.004
`TrovaPerCorsoFallito`	295.39 ± 2.88	0.003 ± 0.000	0.030 ± 0.007
`TrovaPerMatricola`	1084.34 ± 4.44	0.001 ± 0.000	0.025 ± 0.000
`TrovaPerMatricolaFallito`	1090.66 ± 4.20	0.001 ± 0.000	0.026 ± 0.001

Tabella 7.5: Dettaglio Benchmark JMH - Categoria Docente (Baseline)

Benchmark	**Throughput (ops/us)**	**Avg Time (us/op)**	**Sample Time (us/op)**
`aggiungiMessaggio`	0.22 ± 0.01	4.558 ± 0.161	6.458 ± 2.500
`rimuoviMessaggio`	0.22 ± 0.01	4.486 ± 0.103	6.184 ± 2.432
`trovaMessaggio`	0.22 ± 0.01	4.583 ± 0.110	7.371 ± 2.938
`trovaTutti`	0.22 ± 0.01	4.460 ± 0.186	6.173 ± 2.401

Tabella 7.6: Dettaglio Benchmark JMH - Categoria Messaggio (Baseline)

Benchmark	Throughput (ops/us)	Avg Time (us/op)	Sample Time (us/op)
`aggiungiTopic`	0.23 ± 0.00	4.511 ± 0.037	6.559 ± 2.595
`rimuoviTopic`	0.22 ± 0.00	4.542 ± 0.107	5.710 ± 2.111
`trovaCorso`	0.22 ± 0.01	4.575 ± 0.141	5.944 ± 2.262
`trovaCorsoLaurea`	0.22 ± 0.01	4.602 ± 0.152	6.547 ± 2.568
`trovaId`	0.22 ± 0.01	4.638 ± 0.044	6.130 ± 2.337
`trovaNome`	0.22 ± 0.01	4.578 ± 0.112	6.215 ± 2.419
`trovaTutti`	0.22 ± 0.01	4.519 ± 0.165	6.049 ± 2.352

Tabella 7.7: Dettaglio Benchmark JMH - Categoria Topic (Baseline)

Le operazioni dell'Utente mostrano un throughput elevato ed indicano elevata ottimizzazione dell'autenticazione nella baseline. Le query su docente presentano il throughput più alto del sistema, suggerendo efficacia degli indici database su questa entità. Messaggio e Topic hanno, presumibilmente, complessità dovute alle foreign key e relazioni con altre tabelle, con punti di osservazione critici. Tutte le altre hanno operazioni complesse o query meno ottimizzate. I risultati rappresentano un riferimento quantitativo per il refactoring, validando che non introduca regressioni prestazionali sulle operazioni critiche.

7.4.6 System Performance Testing

Rationale

Se JMH è stato utile per la visione delle criticità intra-unità, predisponendo il sistema ad analizzare i services (i.e., Service Layer Pattern Classes per maggiorare la complessità e sicurezza della logica computazionale dei DAO), JMeter sarà utile per comprendere le performance, dal punto di vista sia energetico, sia di banda, dell'intero sistema in punti critici. Apache JMeter simula scenari realistici con carico HTTP concorrente, includendo overhead di rete, servlet container e connection pooling. Esso permette di validare il comportamento del sistema sotto carico anomalo, aspetto critico per i goal di Software Dependability. Inoltre, è ottimo per reporting, data la generazione automatica di dashboard html e csv con alta explainability.

JMeter Configuration

Apache JMeter v5.6.3 permette di organizzare dei test plan in base allo scope e all'ambiente che si vuole eseguire o le connessioni HTTP desiderate. Ogni Test Plan è formato dallo scenario di questo tipo:

- **Thread Group**: Gruppo di thread virtuali che simula utenti concorrenti;
- **HTTP Request Sampler**: Definisce le richeiste HTTP (con eventuali attributi, utili nelle POST);
- **Listeners**: Aggregatori di risultati, come summary report, response time graph, aggregate report, etc.;
- **Assertions**: Validatori di risposta (non utilizzati in questo caso).

Tutti i test JMeter sono stati eseguiti verso il seguente ambiente:

- **Base URL**: `localhost`
- **Port**: `8080`
- **Context Path**: `/UniClass-Dependability`
- **Target Server**: Apache TomEE 10 in Docker Container
- **Protocol**: HTTP/1.1

Test Plan Configurations

Sono stati implementati quattro Test Plan distinti, ciascuno progettato per validare un aspetto specifico delle performance e della resilienza del sistema.

1. Load Test Questo test stabilisce il comportamento del sistema sotto carico discretamente normale e sostenuto, simulando l'utilizzo tipico durante l'orario di lezione.

Parametro	**Valore**
Objective	Determinare throughput e latenza in condizioni operative sotto un carico irrisorio
Thread Count	50 utenti virtuali concorrenti
Ramp-Up Period	10 secondi (5 utenti/secondo)
Loop Count	10 iterazioni per thread
Duration (approx)	∼20–30 secondi
HTTP Requests	`GET /Home` `GET /ChatBot.jsp`
Assertions	Response Code = 200 su Homepage
Rationale	Simula navigazione standard (homepage + feature chatbot). Rappresenta lo scenario di utilizzo più frequente del sistema.

Tabella 7.8: Configurazione Test Plan: Load Test

2. Spike Test (General Pages) Test progettato per validare la capacità del sistema di gestire picchi improvvisi di traffico, come quelli che si verificano solitamente all'apertura delle iscrizioni ai corsi.

Parametro	Valore
Objective	Verificare resilienza a spike di traffico su endpoint pubblici
Thread Count	2000 utenti virtuali concorrenti
Ramp-Up Period	10 secondi (200 utenti/secondo)
Duration	120 secondi (2 minuti)

HTTP Requests	`GET /Home`
	`GET /ChatBot.jsp`
	`GET /mappa.jsp`

Think Time	100 ms tra richieste consecutive
Assertions	Nessuna (focus su error rate e recovery)
Rationale	Simula un evento di traffico anomalo (es. annuncio di nuove funzionalità). Valida che il sistema non entri in stato di errore permanente.

Tabella 7.9: Configurazione Test Plan: Spike Test (General)

3. Authentication Spike Test Test focalizzato sulla funzionalità critica di autenticazione, simulando picchi simultanei di login. Scenario realistico e possibile durante l'inizio delle lezioni di ogni semestre.

Parametro	Valore
Objective	Validare performance e resilienza del sistema di login sotto stress
Thread Count	500 utenti virtuali concorrenti
Ramp-Up Period	5 secondi (100 utenti/secondo)
Duration	60 secondi (1 minuto)
HTTP Requests	`GET /Login.jsp` `POST /Login` (credenziali valide) `POST /Login` (credenziali invalide)
Test Credentials	Email: `m.rossi1@studenti.unisa.it` Password: `Password123!` + credenziali errate per test fallimento
Think Time	50 ms tra richieste consecutive
Assertions	Nessuna (focus su latenza e disponibilità)
Rationale	L'autenticazione è impattata direttamente dal refactoring (modello `Utente`). Questo test valida che le modifiche alla gerarchia delle classi non abbiano degradato le performance di login.

Tabella 7.10: Configurazione Test Plan: Authentication Spike Test

4. Stress Test Test incrementale progettato per identificare il punto di saturazione del sistema con un massimo carico sostenibile visionato poco prima della degradazione critica.

Parametro	Valore
Objective	Identificare il breaking point del sistema e il massimo throughput sostenibile
Thread Count	500 utenti virtuali concorrenti
Ramp-Up Period	300 secondi (5 minuti) - carico incrementale graduale
Duration	360 secondi (6 minuti totali)

HTTP Requests	`GET /Home`
	`GET /ChatBot.jsp`
	`GET /mappa.jsp`

Think Time	300 ms tra richieste consecutive
Assertions	Response Code = 200 su Homepage
Rationale	Il carico viene applicato gradualmente per osservare il comportamento del sistema man mano che ci si avvicina alla saturazione. Permette di identificare il punto esatto in cui error rate o latenza superano soglie accettabili.

Tabella 7.11: Configurazione Test Plan: Stress Test

Risultati dei Test JMeter

I seguenti risultati sono stati ottenuti dalla triplice esecuzione dei quattro Test Plan JMeter sulla baseline del sistema. Tutte le metriche sono aggregate dai file `.csv` generati automaticamente da JMeter Listeners.

Endpoint	Samples	Avg (ms)	Min (ms)	Max (ms)	Error %	Throughput (req/s)
Homepage	1000	13	3	86	0.000%	18.89
Chat	1000	3	0	52	0.000%	18.89
TOTAL	**2000**	**8**	**0**	**86**	**0.000%**	**37.56**

Tabella 7.12: Risultati JMeter - Load Test

Load Testing Results La media è di 13 ms ed un picco di 86 ms, tempi ottimali per una pagina con contenuti dinamici, così come la latenza del ChatBot ed un throughput accettabile.

Endpoint	Samples	Avg (ms)	Min (ms)	Max (ms)	Error %	Throughput (req/s)
Homepage	16401	13949	4	32166	0.000%	48.79
ChatBot	14224	11349	1	26912	0.000%	42.37
Mappa	12996	7932	0	26888	0.000%	38.73
TOTAL	**43621**	**11308**	**0**	**32166**	**0.000%**	**129.75**

Tabella 7.13: Risultati JMeter - Spike Test

Spike Testing Results La Homepage ha una degradazione prestazionale molto pesante per via dello startup. I picchi raggiungono troppi secondi (32), inaccettabili. Il throughput è ottimo, nonostante l'insieme delle utenze attive, ma con latenze molto alte che compensano.

Endpoint	Samples	Avg (ms)	Min (ms)	Max (ms)	Error %	Throughput (req/s)
Login Page (GET)	18871	182	1	2795	0.000%	314.36
Login POST (Valid)	18784	595	2	3461	0.000%	312.23
Login POST (Invalid)	18565	599	6	3492	0.000%	309.21
TOTAL	**56220**	**458**	**1**	**3492**	**0.000%**	**933.67**

Tabella 7.14: Risultati JMeter - Authentication Spike Test

Authentication Spike Testing Results La latenza POST Login è molto alta rispetto alla medesima di metodo d'unità visionata nei Microbenchmark; la Latenza GET è ottimale; c'è un collo di bottiglia nella validazione delle credenziali, più presente nella query che nella validazioen logica.

Endpoint	Samples	Avg (ms)	Min (ms)	Max (ms)	Error %	Throughput (req/s)
Homepage	16630	4580	2	16196	0.000%	45.51
ChatBot	16171	649	0	9485	0.000%	44.31
Mappa	16130	341	0	10566	0.000%	45.26
TOTAL	**48931**	**1883**	**0**	**16196**	**0.000%**	**133.91**

Tabella 7.15: Risultati JMeter - Stress Test

Stress Testing Results La degradazione è anche qui significativa per la latenza Homepage. Il sistema riesce ad adattarsi meglio rispetto allo spike improvviso, avendo latenze decisamente inferiori. Il throughput è ben distribuito e con 500 utenti il sistema degrada ma non collassa. Si può determinare che la capacità massima del sistema (vera e propria) è maggore di 500 utenti.

Conclusioni Generali

Il sistema mantiene un error rate minimo in ogni scenario, disponendo alta disponibilità ed affidabilità, indipendentemente dalla criticità dello scenario preposto. Vi è una degradazione prestazionale critica per via dell'aumento della latenza improvviso, rendendo il sistema inutilizzabile durante picchi di traffico.
I colli di bottiglia identificati riguardano il DB Connection Pool, un Thread Pool di Tomcat limitato sul numero di thread worker disponibili e query JPA troppo complesse, creando overhead dei JOIN amplificato dalla concorrenza.
Data la complessità delle query e di Tomcat, si modificherà l'implementazione e l'architettura dove possibile, sulla base dell'idea della manutenzione correttiva e perfettiva, avendo come obiettivo il miglioramento dell'esperienza utente.

Il Validation Target per le analisi Post-Refactoring riguardano un decremento della latenza di almeno 20% o un eventuale gestore di connessioni concorrenti, prevenendo in ogni caso il DDoS Attack.

Capitolo 8

Item Pass/Fail Criteria

L'intervento di manutenzione sarà considerato concluso con successo solo al soddisfacimento dei seguenti criteri:

Categoria	Criterio di Successo (Pass)
Regressione Funzionale	100% dei Test Case critici (Login, Core Business) passati. Nessun bug di severità "Blocker" o "Critical".
Performance (Startup)	$T_{Post} < T_{Baseline}$ (Il tempo di startup deve essere inferiore alla baseline di 18ms).
Performance (Throughput)	$Ops_{Post} \geq Ops_{Baseline}$ (Il throughput delle query utente non deve degradare oltre il 2%).
Qualità del Codice	• Code Coverage $\geq 80\%$ sulle classi modificate. • Mutation Score $\geq 50\%$ (come da baseline precedente). • 0 Vulnerabilità di sicurezza (SonarCloud).

Tabella 8.1: Criteri di accettazione

Capitolo 9

Test Deliverables

Al termine del processo verranno rilasciati i seguenti artefatti:

- **TCS:** Specifica dei casi di test aggiornata e sotto la supervisione di Version Control.
- **TER:** Report riassuntivo con confronto metriche Pre/Post intervento.

Capitolo 10

Assessment: valutazione della Test Suite attuale

Questa sezione analizza lo stato della test suite presente nel repository `uniclass-dependability` prima dell'applicazione della Change Request. L'analisi si basa sul documento *"UniClass - Report SwD"* e sull'ispezione del codice sorgente.

10.1 Stato Attuale - "As Is"

Il progetto parte da una situazione di **elevata affidabilità**, frutto di un precedente intervento di *Software Dependability*.

Metrica	Valore Baseline	Valutazione
Framework di Test	JUnit 5, Mockito, Arquillian	Moderno e Adeguato
Code Coverage	Elevata (Package critici coperti)	Conforme
Mutation Score (PiTest)	$\geq 50\%$	Robusto
Analisi Statica (Sonar)	0 Vulnerabilità	Sicuro
Performance Testing	JMH Presente	Pronto all'uso
Manutenibilità Test	Uso di Mock Manuali	Rischio Medio

Tabella 10.1: Assessment della Baseline (Fonte: Report SwD)

10.2 Analisi dei Rischi sulla Test Suite (Impact Analysis)

Nonostante l'alta qualità della baseline, la Change Request (Refactoring da Ereditarietà a Composizione) avrà un impatto **distruttivo** sulla suite di test unitari esistente.

- **Unit Tests (Model):** `StudenteTest`, `DocenteTest`.

  - **Stato: Critico**.
  - **Motivo:** Questi test istanziano classi concrete (`new Studente()`) che verranno rimosse o modificate drasticamente.
  - **Azione:** Sarà necessario riscrivere questi test per utilizzare la nuova classe `Utente` con l'iniezione della corretta *Strategy* o *Role*.

- **Integration Tests (DAO):** `StudenteDAO`, `UtenteDAO`.
  - **Stato: Attenzione**.
  - **Motivo:** Le query JPQL nei test potrebbero fallire se il nome delle entità o dei campi mappati cambia.
  - **Azione:** Verifica e adattamento delle query nei test di integrazione Arquillian.

- **Performance Benchmarks (JMH):**
  - **Stato: Stabile**.
  - **Motivo:** Testano l'interfaccia pubblica dei Service, che dovrebbe rimanere invariata. Saranno fondamentali come "Oracolo" per verificare il successo dell'ottimizzazione (Target: battere i 18ms attuali).

10.3 Conclusione Assessment

La test suite attuale è **eccellente come punto di partenza** ma non può essere utilizzata "così com'è" durante il refactoring. La strategia non prevede la creazione di nuovi test da zero, bensì un'attività di **Refactoring dei Test** parallela al refactoring del codice, utilizzando i Benchmark JMH e i Test di Sistema (Selenium & JMeter) come rete di sicurezza (Safety Net) per garantire che le funzionalità esterne non regrediscano mentre si ristruttura il codice interno.