ODD
OBJECT DESIGN DOCUMENT

ModuLink

Riferimento	NC08_ODD_ver.1.1
Versione	1.1
Data	10/12/2025
Destinatario	Studenti di Ingegneria del Software 2025/26
Presentato da	Buzi Arjel, Carpentieri Daniele, Chikviladze Aleksandre, Cito Roberto.
Codice Gruppo	NC08
Approvato da	

Revision History

Data	Versione	Descrizione	Autori
10/12/2025	0.1	Prima stesura	Roberto
11/12/2025	0.2	Sezioni 1-2-3	Roberto, Arjel
16/12/2025	0.3	Design Pattern	Tutti
20/12/2025	0.4	Revisione	Tutti
19/01/2026	0.5	Aggiornamento descrizioni Design Pattern Post Suggerimenti	Daniele
20/01/2026	1.0	Revisione finale	Aleksandre
03/02/2026	1.1	Correzione Post-Esame	Roberto

Team members

Nome	Ruolo nel progetto	Acronimo	Informazioni di contatto
Roberto Cito	Team Member	RC	r.cito@studenti.unisa.it
Daniele Carpentieri	Team Member	DC	d.carpentieri8@studenti.unisa.it
Aleksandre Chikviladze	Team Member	AC	a.chikviladze@studenti.unisa.it
Arjel Buzi	Team Member	AB	a.buzi@studenti.unisa.it

Glossario

Termine	Descrizione
Package	Un **contenitore logico e strutturale** utilizzato per raggruppare classi, interfacce e componenti correlati. Il suo scopo principale è organizzare il codice in modo gerarchico e modulare, prevenendo conflitti di nomi e semplificando la manutenzione del progetto.
Design Pattern	Uno **schema progettuale collaudato** che offre una risoluzione standardizzata a problematiche frequenti nella programmazione. Non si tratta di codice pronto all'uso, bensì di un modello concettuale o una *best practice* da adattare per risolvere in modo efficiente specifici contesti architetturali.
Lower Camel Case	Uno stile di notazione per identificatori (come variabili o metodi) in cui la parola iniziale comincia con la **lettera minuscola**, mentre ogni parola successiva unita alla prima inizia con la maiuscola, senza spazi intermedi (es. ilMioMetodo).
Upper Camel Case	Una convenzione di scrittura spesso nota anche come *PascalCase*, in cui **ogni parola** che compone il nome, inclusa la primissima, inizia con una **lettera maiuscola**. È lo standard *de facto* per la denominazione di Classi e Interfacce in numerosi linguaggi di programmazione.
JavaDoc	Il generatore di documentazione standard per il linguaggio Java. Questa utility analizza i commenti formattati in modo specifico all'interno del codice sorgente per produrre automaticamente una **documentazione tecnica** (solitamente in formato HTML) che descrive la struttura e le funzionalità di classi e metodi.

Sommario

Revision History 2
Team members 3
Glossario 4
1 Introduzione 6
1.1 Obiettivo del Sistema 6
1.2 Riferimenti 6
2 Obiettivi dell'Object Design 7
3 Linee Guida 8
4 Struttura del Progetto 9
4.1 Package 9
5 Design Pattern 11
5.1 Facade 11
5.1.1 Implementazione in ModuLink: 11
5.2 Bridge 13
5.2.1 Implementazione in ModuLink: 13

1 Introduzione

1.1 Obiettivo del Sistema

l'obiettivo è quello di fornire alle aziende uno strumento centralizzato e modulare capace di semplificare la gestione delle attività interne e delle risorse digitali. Il sistema è progettato per supportare:

- la gestione dei moduli aziendali
- l'organizzazione strutturata degli utenti e dei ruoli
- la gestione operativa di task, calendari ed eventi
- l'amministrazione di risorse e prodotti tramite il modulo magazzino
- l'integrazione di moduli personalizzati per esigenze specifiche dell'azienda

1.2 Riferimenti

Di seguito sono riportati i riferimenti a risorse che possono agevolare la lettura del documento, o fornire maggiori informazioni.

Nome Riferimento	Riferimento
Libro	"Object-Oriented Software Engineering: Conquering Complex and Changing Systems", terza edizione, Bernd Bruegge & Allen Dutoit, 2014.
SOW	Statement Of Work (SOW) Chiuso
RAD	Requirement Analysis Document Chiuso
SDD	System Desing Document Chiuso
TPD	Test Plan Document Chiuso

2 Obiettivi dell'Object Design

Regola	Descrizione
Riusabilità	Al fine di massimizzare il riutilizzo delle componenti software e minimizzare la ridondanza logica, il sistema implementa paradigmi architetturali fondati sull'**ereditarietà** e sull'applicazione sistematica di **Design Pattern** standardizzati. Tale metodologia garantisce l'estensibilità modulare delle funzionalità, senza la necessità di refactoring o riscritture del codice preesistente.
Robustezza	L'affidabilità del sistema e la sua capacità di operare anche in condizioni impreviste sono assicurate da una gestione puntuale delle **exception**. Attraverso il controllo e la risoluzione degli errori in *run-time*, si previene l'interruzione del servizio garantendo una continuità operativa stabile.
Incapsulamento	L'architettura mira a nascondere la complessità interna e i dettagli logici attraverso l'impiego di **interfacce**. Questo favorisce un basso accoppiamento, poiché le componenti interagiscono basandosi su contratti definiti (le interfacce) piuttosto che sulla conoscenza diretta delle implementazioni sottostanti.

3 Linee Guida

Al fine di assicurare un'elevata manutenibilità e uniformità del codice sorgente, il team di sviluppo è tenuto a rispettare le seguenti direttive stilistiche. L'adesione a queste norme favorisce una lettura fluida degli elaborati e snellisce le procedure di code review.

Regola	Descrizione
Denominazione Classi	Gli identificatori delle classi devono esplicitarne chiaramente lo scopo e seguire il formato **Upper Camel Case**. È obbligatorio l'uso di suffissi specifici in base al ruolo architetturale: le classi di controllo devono terminare con **"Control"**, le implementazioni della logica di business con **"ServiceImpl"** e gli oggetti di dominio (modelli dati) con **"Entity"**.
Denominazione Interfacce	Le interfacce, come le classi, adottano l'**Upper Camel Case** e devono avere nomi auto-esplicativi. Per le interfacce di servizio il suffisso richiesto è **"Service"**, mentre per i componenti dedicati alla persistenza e accesso ai dati si dovrà utilizzare il suffisso **"Repository"**.
Lingua delle Eccezioni	La nomenclatura per tutte le classi di eccezione personalizzate deve essere rigorosamente in **lingua inglese**.
Formato Variabili	Per gli identificatori di variabile (campi o variabili locali) si applica esclusivamente la notazione **lower camel case**.
Dichiarazione Metodi (Throws)	Al fine di garantire la leggibilità della firma dei metodi, qualora venga sollevata una **singola eccezione**, questa deve essere dichiarata sulla medesima riga della firma. In presenza di **eccezioni multiple**, la clausola ***throws*** va formattata disponendo l'elenco delle stesse alla riga successiva.

4 Struttura del Progetto

4.1 Package

Di seguito è riportata la suddivisione del sistema in packages. Tale suddivisione segue l'architettura del sistema la suddivisione di quest'ultimo in sottosistemi, entrambe descritte all'interno dei documenti precedenti. Innanzitutto, dovrà essere definito un package per ogni livello dell'architettura. Per cui avremo:

```
+---src
|  +---main
|  |  +---java
|  |  |  \---com
|  |  |     \---modulink
|  |  |        +---Controller
|  |  |        |  +---AdminModules
|  |  |        |  |  +---Manage
|  |  |        |  |  +---News
|  |  |        |  |  \---Support
|  |  |        |  +---Dashboard
|  |  |        |  +---EditUser
|  |  |        |  +---HomePage
|  |  |        |  +---Login
|  |  |        |  +---Register
|  |  |        |  \---UserModules
|  |  |        |     +---GDE
|  |  |        |     +---GDM
|  |  |        |     +---GDR
|  |  |        |     +---GDU
|  |  |        |     +---GMA
|  |  |        |     |  +---Role
|  |  |        |     |  \---Store
|  |  |        |     +---GRU
|  |  |        |     \---GTM
```

```
| | |       +---DatabasePopulator
| | |       \---Model
| | |         +---Azienda
| | |         +---Email
| | |         +---Eventi
| | |         +---Modulo
| | |         +---News
| | |         +---OTP
| | |         +---Prodotto
| | |         +---Relazioni
| | |         |  +---Assegnazione
| | |         |  +---Associazione
| | |         |  +---Attivazione
| | |         |  +---Partecipazione
| | |         |  \---Pertinenza
| | |         +---Ruolo
| | |         +---SupportForm
| | |         +---Task
| | |         \---Utente
| \---test
|     \---java
|       \---com
|         \---modulink
|           \---Controller
|             +---Register
|             \---UserModules
|               +---GDE
|               +---GDM
|               \---GTM
```

5 Design Pattern

La progettazione dell'architettura software del sistema si fonda sull'adozione di soluzioni collaudate per garantire la manutenibilità, la scalabilità e la chiarezza del codice. L'analisi delle problematiche strutturali e comportamentali emerse durante la fase di design ha guidato la selezione di specifici Design Pattern, utilizzati come standard per risolvere ricorrenze architetturali comuni.

Per rispondere alle esigenze di **flessibilità** nell'interazione tra moduli e per garantire una **indipendenza evolutiva** tra le interfacce e le loro implementazioni, abbiamo integrato nel progetto due pattern strutturali fondamentali: il **Facade** e il **Bridge**.

Le motivazioni tecniche che hanno portato a queste scelte sono dettagliate di seguito:

5.1 Facade

Un Facade è un punto di accesso unico e semplificato verso un sottosistema complesso.

In **ModuLink**, il sottosistema complesso è composto da:

- Repository (query e persistenza)
- altre Service (ruoli, permessi, relazioni, ecc.)
- transazioni e caching
- regole di business (validazioni “di dominio”, coerenza tenant, ecc.)

I controller (client) non devono conoscere questi dettagli: chiamano metodi “ad alto livello” su un Service.

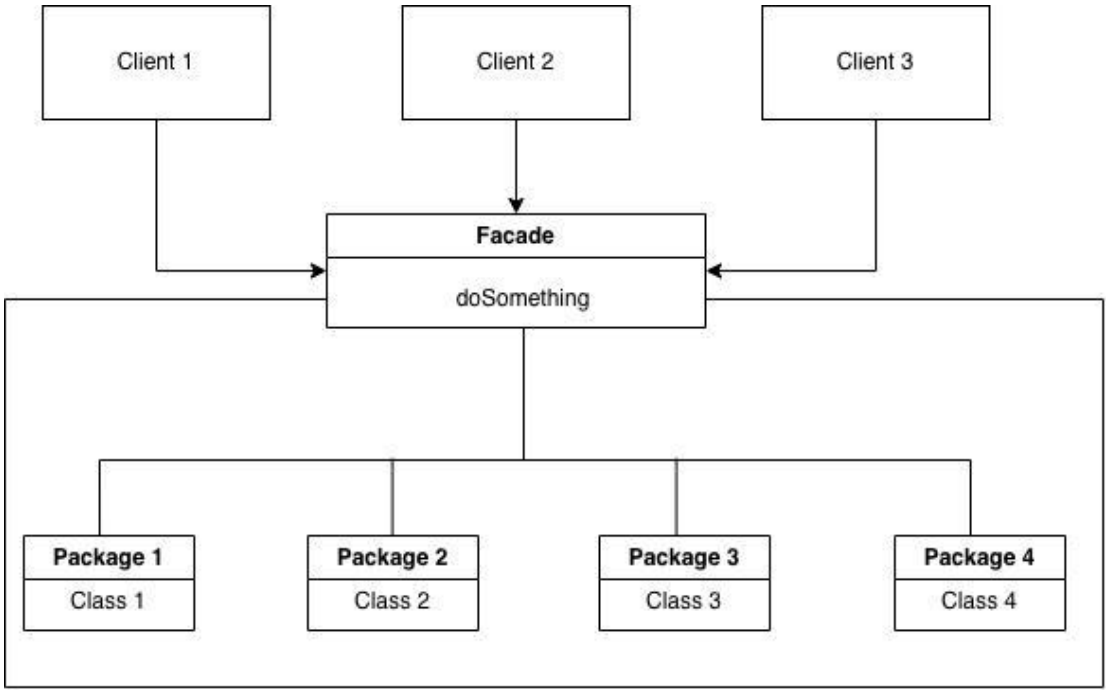
Perché l’abbiamo implementato?

1. Riduce l’accoppiamento: i controller non dipendono da repository multipli o dalla struttura interna delle relazioni.
2. Evita duplicazione: la stessa sequenza di passi (es. controlli + salvataggi + pulizia) sta in un punto solo.
3. Centralizza regole e sicurezza: la logica rimane coerente e modificabile senza cambiare 10 controller.
4. Gestisce aspetti trasversali (bene nel tuo codice): @Transactional, @Cacheable, @CacheEvict.

5.1.1 Implementazione in ModuLink:

Il pattern Facade è stato concretamente implementato tramite i service di dominio (es. ModuloService, AttivazioneService, CustomUserDetailsService, ecc.).

Queste classi fungono da punto di accesso semplificato per i Controller per interagire con sottosistemi complessi (repository + relazioni + caching + transazioni), senza esporre la struttura interna.



5.2 Bridge

Il pattern Bridge è stato applicato per disaccoppiare il sistema di Notifiche dalla logica di invio specifica.

5.2.1 Implementazione in ModuLink:

- **Abstraction:**
  ModuloController è la classe base astratta che espone un'API comune ai moduli:
  - tiene l'id del modulo
  - espone isAccessibleModulo(...) (logica comune)
  - impone disinstallaModulo(...) (contratto che ogni modulo deve implementare)
- **Implementor:** ModuloController non sa come si decide l'accesso ad un modulo: lo delega a ModuloService, che incapsula query/relazioni/caching.
- **Refined Abstractions:**
  i controller dei singoli moduli (es. GTMController, GDMController, …), ed ogni modulo:
  - "configura" l'astrazione passando l'ID modulo al super(...)
  - usa isAccessibleModulo(...) prima di eseguire la logica specifica
  - implementa disinstallaModulo(...) con la pulizia dati del proprio dominio

In questo modo, il sistema può evolvere aggiungendo nuovi tipi di notifiche (es. *PushNotification*) o nuovi metodi di invio senza dover modificare la gerarchia delle classi esistenti.