Requisiti Funzionali del Sistema

Il sistema Dante è composto da due componenti:

Client

1. **Pre-elaborazione del testo utente**
   - Divisione in termini (tokenizzazione)
   - Costruzione di una struttura ad albero (rappresentazione semantica)
   - Compressione con algoritmo proprietario
2. **Invio al server**
   - Comunicazione asincrona
   - Invio dati compressi

Server

1. **Ricezione e decompressione**
   - Ricezione dati compressi
   - Decompressione algoritmo proprietario
2. **Interpretazione**
   - Scorrimento **non standard** della struttura ad albero
   - Generazione risposta
3. **Risposta al client**
   - Invio asincrono della risposta

Mappatura Pattern → Requisiti

1. COMPOSITE PATTERN (Strutturale)

Requisito coperto: *"costruendo una struttura ad albero che rappresenta semanticamente il testo"*

Scopo secondo Cardin:

> Comporre oggetti in strutture ad albero, che rappresentano gerarchie intero-parte, e consentire ai client di trattare oggetti singoli e compositi in modo uniforme.

Applicazione nel sistema:

- Il testo viene diviso in termini (tokens)
- Questi termini formano una struttura gerarchica ad albero

- I nodi dell'albero possono essere:
  - **Foglie** (`Term`): termini singoli atomici
  - **Compositi** (`CompositeTerm`): aggregazioni semantiche di sottostrutture

Giustificazione:

- La "rappresentazione semantica" implica una struttura gerarchica
- L'albero deve permettere operazioni uniformi su nodi semplici e composti
- Il Composite è l'unico pattern strutturale del corso che gestisce strutture ad albero

Classi coinvolte:

```
TermComponent (Component astratto)
├── Term (Leaf)
└── CompositeTerm (Composite)
```

---

2. STRATEGY PATTERN (Comportamentale)

Requisito coperto: *"effettuando una compressione degli stessi con un algoritmo proprietario"*

Scopo secondo Cardin:

> Definisce una famiglia di algoritmi, rendendoli interscambiabili e indipendenti dal client.

Applicazione nel sistema:

- L'algoritmo di compressione è **proprietario** (quindi specifico e sostituibile)
- Sia il client che il server devono usare lo stesso algoritmo
- L'algoritmo deve essere separato dalla logica di client/server

Giustificazione:

- "Algoritmo proprietario" suggerisce un algoritmo intercambiabile
- Il testo specifica che l'algoritmo è **proprietario**, non standard
- Strategy permette di variare l'algoritmo senza modificare client/server

Classi coinvolte:

```
CompressionStrategy (Strategy interface)
├── ProprietaryCompression (ConcreteStrategyA)
└── AlternativeCompression (ConcreteStrategyB)

Client e Server mantengono un riferimento a CompressionStrategy
```

3. ITERATOR PATTERN (Comportamentale)

Requisito coperto: *"usando un algoritmo la cui implementazione richiede lo scorrimento della struttura ad albero in modo non standard"*

Scopo secondo Cardin:

> Fornisce l'accesso sequenziale agli elementi di un aggregato senza esporre l'implementazione dell'aggregato. Supporto a variazioni nelle politiche di attraversamento.

Applicazione nel sistema:

- Il server deve interpretare l'albero
- Lo scorrimento è esplicitamente **non standard**
- L'algoritmo di attraversamento deve essere separato dalla struttura

Giustificazione:

- "Scorrimento non standard" indica una modalità di attraversamento personalizzata
- Iterator permette diverse politiche di attraversamento
- Separazione tra struttura (Composite) e attraversamento (Iterator)

Classi coinvolte:

```
TreeIterator (Iterator interface)
└── NonStandardTreeIterator (ConcreteIterator)

TermComponent implementa accept(TreeIterator) per permettere
l'attraversamento
```

Pattern correlato: Il design è simile al Visitor, ma nel corso di Cardin il Visitor non è trattato, quindi si usa l'Iterator per l'attraversamento personalizzato.

---

4. COMMAND PATTERN (Comportamentale)

Requisito coperto: *"il client invia le informazioni compresse al server"* + *"comunicazione è asincrona"*

Scopo secondo Cardin:

> Incapsulare una richiesta in un oggetto, cosicché i client siano indipendenti dalle richieste. Specificare, accodare ed eseguire richieste molteplici volte.

Applicabilità dal corso:

> "Definire, eseguire e accodare richieste in un oggetto. In particolare permette di eseguire operazioni indipendenti in modo asincrono."

Applicazione nel sistema:

- Le richieste client→server devono essere incapsulate
- La comunicazione è **asincrona** (esecuzione non bloccante)
- Il server è il receiver che esegue l'elaborazione

Giustificazione:

- L'applicabilità del Command esplicitamente menziona "modo asincrono"
- Le richieste sono oggetti indipendenti trasmessi via rete
- Il client non sa quando/come il server eseguirà la richiesta

Classi coinvolte:

```
Command (Command interface)
└── ProcessRequestCommand (ConcreteCommand)
    ├── compressedData: byte[]
    └── receiver: DanteServer

RequestInvoker (Invoker)
└── executeAsync() per gestione asincrona

DanteServer (Receiver)
```

---

5. OBSERVER PATTERN (Comportamentale)

Requisito coperto: *"la loro comunicazione è asincrona"* + *"il testo risultante [...] viene ritornato al client"*

Scopo secondo Cardin:

> Definisce una dipendenza di tipo 1..n fra oggetti riflettendo la modifica di un oggetto su quelli che sono a lui relazionati.

Applicazione nel sistema:

- La comunicazione è bidirezionale e asincrona
- Il client deve essere **notificato** quando il server ha completato l'elaborazione
- Il client non sa quando arriverà la risposta (polling inefficiente)

Giustificazione:

- "Comunicazione asincrona" implica notifiche push
- Il client deve reagire quando il server completa l'elaborazione
- Observer è il pattern standard per notifiche asincrone

Classi coinvolte:

```
ResponseObserver (Observer interface)
└── ClientResponseHandler (ConcreteObserver)

DanteServer (Subject)
├── observers: List<ResponseObserver>
└── notify() quando risposta è pronta

DanteClient può registrarsi come observer
```

Integrazione dei Pattern

Flusso Completo

```
1. CLIENT - Pre-elaborazione
   ├─> Tokenizzazione → List<Term>
   ├─> Costruzione albero (COMPOSITE)
   └─> Compressione (STRATEGY)

2. CLIENT → SERVER - Comunicazione Asincrona
   ├─> Incapsulamento richiesta (COMMAND)
   └─> Invio tramite RequestInvoker

3. SERVER - Ricezione
   └─> ProcessRequestCommand.execute()

4. SERVER - Elaborazione
   ├─> Decompressione (STRATEGY)
   ├─> Scorrimento non standard (ITERATOR)
   └─> Generazione risposta

5. SERVER → CLIENT - Notifica Asincrona
   ├─> Server chiama notify() (OBSERVER)
   └─> ClientResponseHandler.update()

6. CLIENT - Ricezione
   └─> Visualizzazione risposta
```

Vantaggi dell'Architettura

1. **Separazione delle responsabilità**
   - Ogni pattern gestisce un aspetto specifico
   - Alta coesione, basso accoppiamento
2. **Estensibilità**
   - Nuovi algoritmi di compressione (Strategy)
   - Nuove modalità di attraversamento (Iterator)
   - Nuovi tipi di comando (Command)
   - Nuovi observer (Observer)
3. **Testabilità**
   - Ogni componente può essere testato in isolamento
   - Mock facili da creare per ogni interfaccia
4. **Manutenibilità**
   - Modifiche localizzate
   - Pattern ben documentati e standard

Pattern Alternative Considerate (e Scartate)

Visitor invece di Iterator

- **Non trattato** nel corso di Cardin (vedi pratica_cardin.pdf)
- Iterator è sufficiente per l'attraversamento personalizzato

Proxy per comunicazione

- Proxy gestisce **accesso** a oggetti, non comunicazione asincrona
- Command + Observer sono più adatti per asincronia

Mediator per Client-Server

- Mediator centralizza comunicazione tra molti oggetti
- Sistema Dante ha solo 2 componenti (client e server)
- Command + Observer sono sufficienti

Template Method invece di Iterator

- Template Method definisce **scheletro** di algoritmo
- Non permette attraversamento personalizzato di strutture dati
- Iterator è specifico per attraversamento collezioni/alberi

Conformità al Corso di Cardin

Tutti i 5 pattern applicati sono:

1. **Presenti** nel materiale del corso:
   - Composite ✓ (Pattern Strutturale)
   - Strategy ✓ (Pattern Comportamentale)
   - Iterator ✓ (Pattern Comportamentale)
   - Command ✓ (Pattern Comportamentale)
   - Observer ✓ (Pattern Comportamentale)
2. **Correttamente applicati** secondo le definizioni di Cardin:
   - Scopo rispettato ✓
   - Struttura conforme ✓
   - Applicabilità appropriata ✓
3. **Giustificati** dai requisiti funzionali:
   - Ogni pattern risolve un requisito specifico ✓
   - Nessun pattern superfluo ✓

Verifica Requisiti → Pattern

Requisito	Pattern	Motivazione
Struttura ad albero semantica	**Composite**	Gerarchia intero-parte
Algoritmo proprietario compressione	**Strategy**	Algoritmo intercambiabile
Scorrimento non standard albero	**Iterator**	Politiche attraversamento
Comunicazione asincrona (invio)	**Command**	Richieste incapsulate asincrone
Comunicazione asincrona (risposta)	**Observer**	Notifiche push asincrone

✓ **Tutti i requisiti funzionali sono coperti dai pattern del corso**.