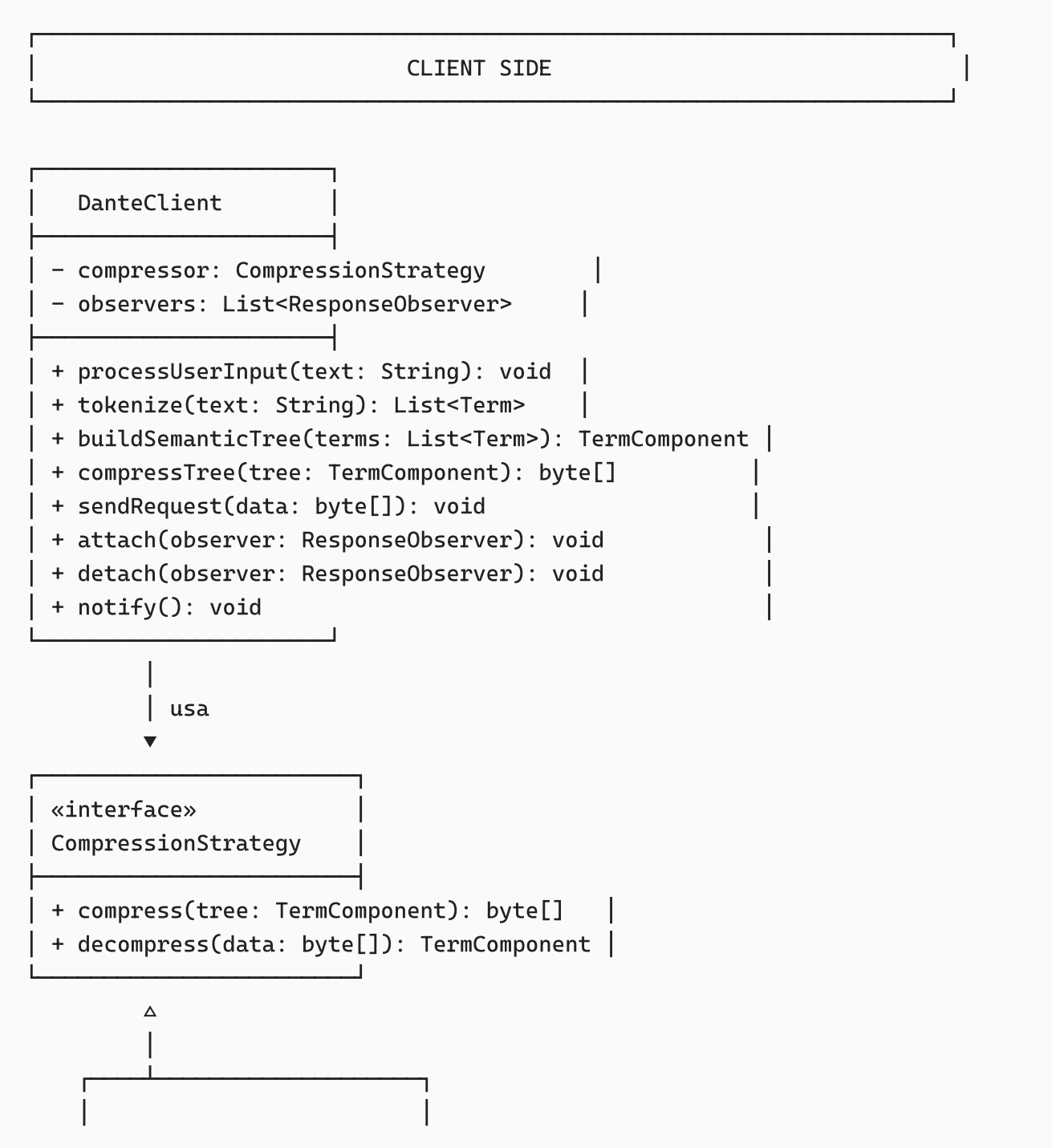
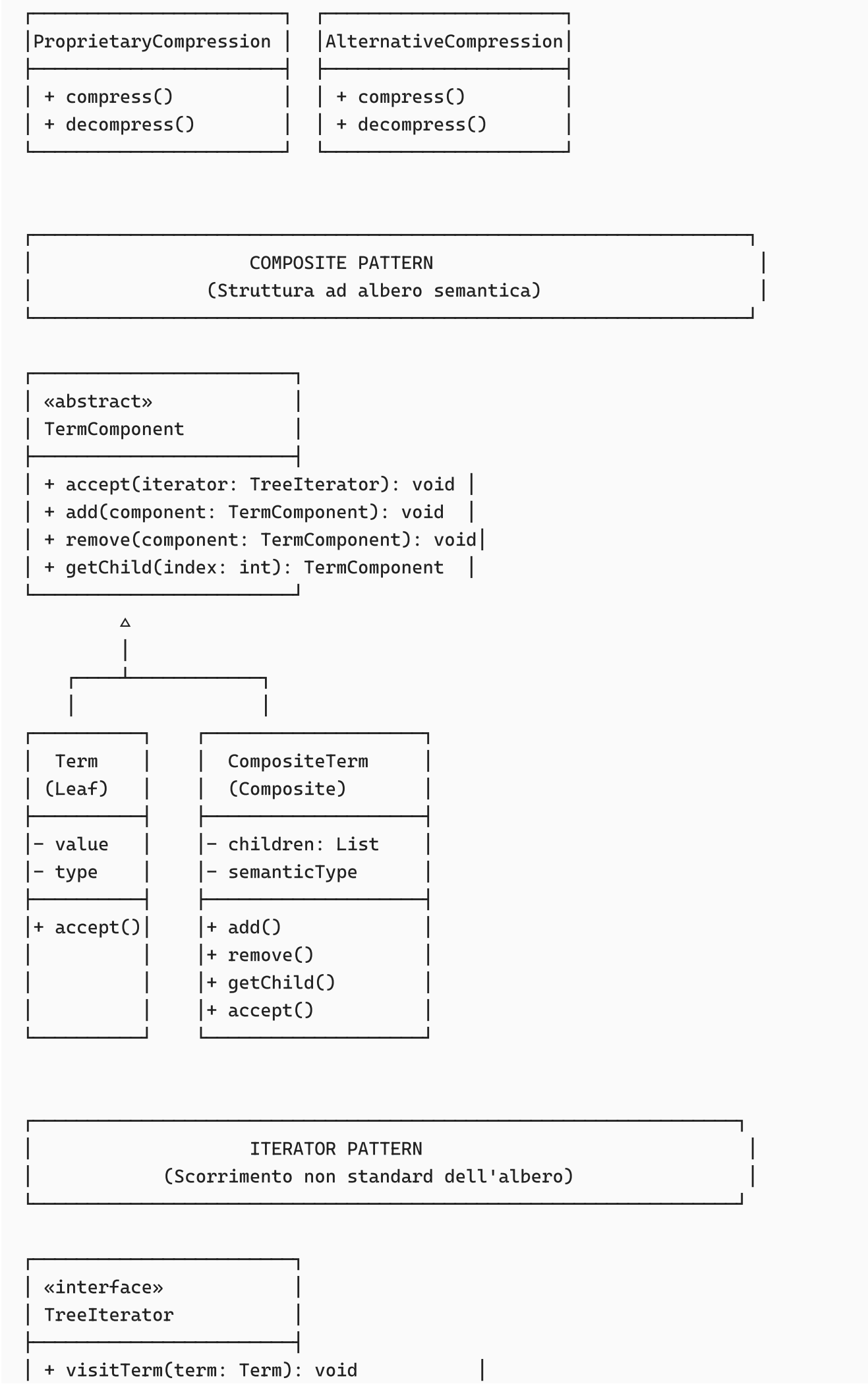
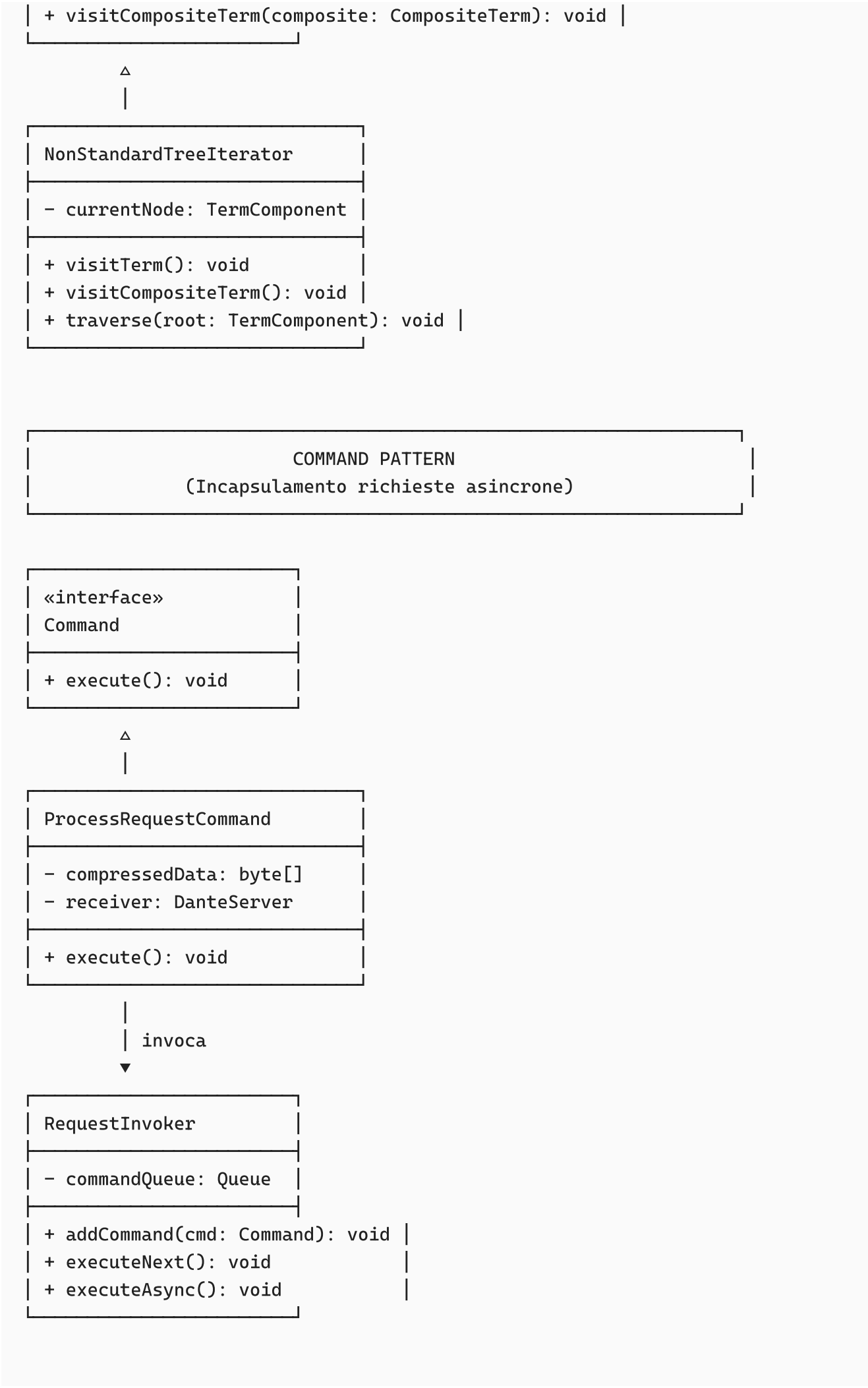
Pattern Applicati

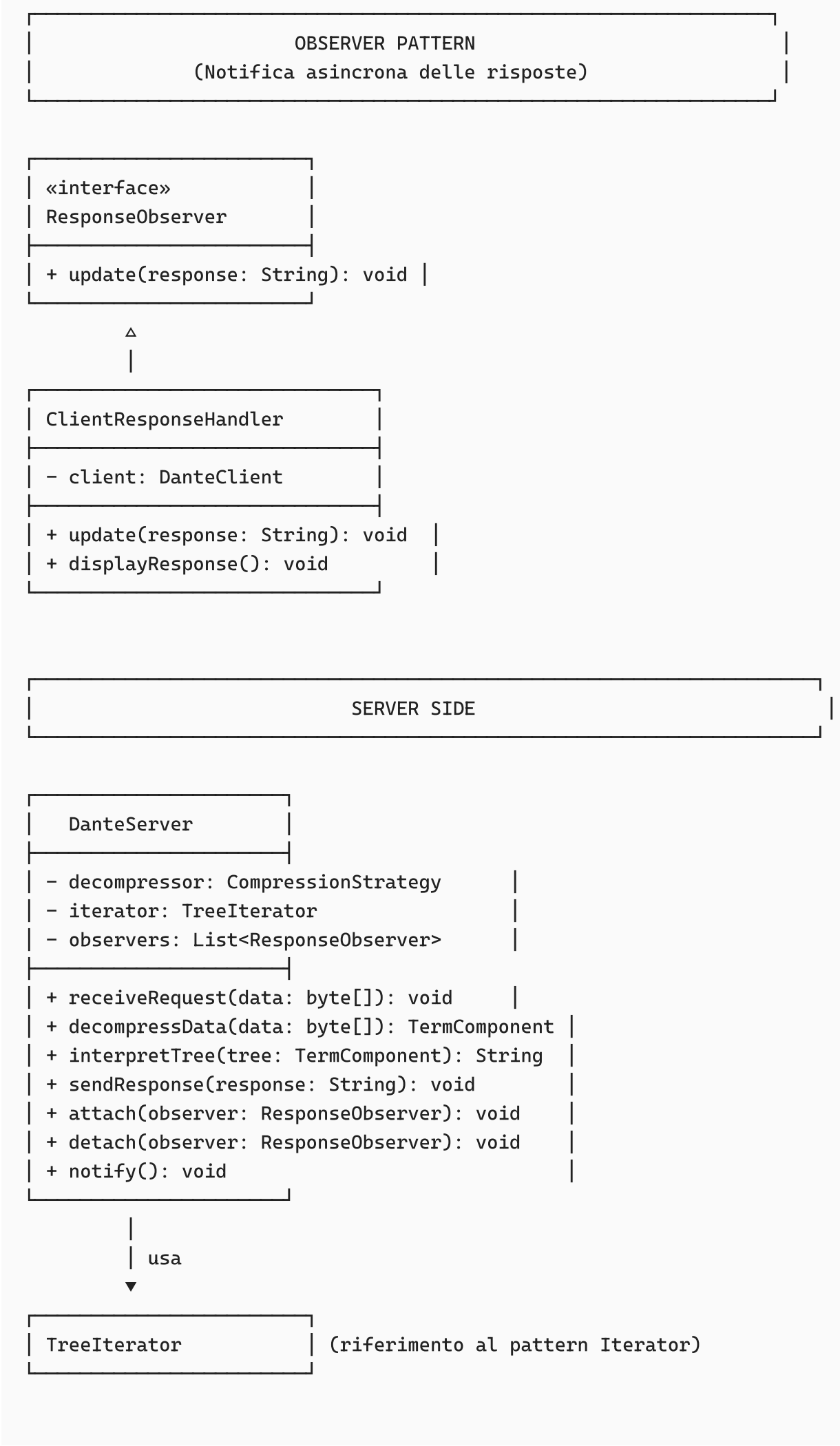
- **Composite**: Struttura ad albero dei termini
- **Strategy**: Algoritmo di compressione intercambiabile
- **Iterator**: Scorrimento non standard dell'albero
- **Command**: Richieste asincrone incapsulate
- **Observer**: Notifica asincrona delle risposte

Struttura delle Classi




ProprietaryCompression
+ compress()
+ decompress()
AlternativeCompression
+ compress()
+ decompress()
COMPOSITE PATTERN
(Struttura ad albero semantica)
«abstract»
TermComponent
+ accept(iterator: TreeIterator): void
+ add(component: TermComponent): void
+ remove(component: TermComponent): void
+ getChild(index: int): TermComponent
Term
(Leaf)
- value
- type
+ accept()
CompositeTerm
(Composite)
- children: List
- semanticType
+ add()
+ remove()
+ getChild()
+ accept()
ITERATOR PATTERN
(Scorrimento non standard dell'albero)
«interface»
TreeIterator
+ visitTerm(term: Term): void


+ visitCompositeTerm(composite: CompositeTerm): void
NonStandardTreeIterator
- currentNode: TermComponent
+ visitTerm(): void
+ visitCompositeTerm(): void
+ traverse(root: TermComponent): void
COMMAND PATTERN
(Incapsulamento richieste asincrone)
«interface»
Command
+ execute(): void
ProcessRequestCommand
- compressedData: byte[]
- receiver: DanteServer
+ execute(): void
invoca
RequestInvoker
- commandQueue: Queue
+ addCommand(cmd: Command): void
+ executeNext(): void
+ executeAsync(): void


OBSERVER PATTERN
(Notifica asincrona delle risposte)
«interface»
ResponseObserver
+ update(response: String): void
ClientResponseHandler
- client: DanteClient
+ update(response: String): void
+ displayResponse(): void
SERVER SIDE
DanteServer
- decompressor: CompressionStrategy
- iterator: TreeIterator
- observers: List<ResponseObserver>
+ receiveRequest(data: byte[]): void
+ decompressData(data: byte[]): TermComponent
+ interpretTree(tree: TermComponent): String
+ sendResponse(response: String): void
+ attach(observer: ResponseObserver): void
+ detach(observer: ResponseObserver): void
+ notify(): void
usa
TreeIterator
(riferimento al pattern Iterator)

```
┌──────────────────────────────────────────────────────────────┐
│                    RELAZIONI PRINCIPALI                      │
└──────────────────────────────────────────────────────────────┘

DanteClient ──────> CompressionStrategy (Strategy Pattern)
DanteClient ──────> TermComponent (Composite Pattern)
DanteClient ──────> ProcessRequestCommand (Command Pattern)
DanteClient ──────> ResponseObserver (Observer Pattern - Subject)

ProcessRequestCommand ──────> DanteServer (Command Pattern - Receiver)

DanteServer ──────> CompressionStrategy (Strategy Pattern)
DanteServer ──────> TreeIterator (Iterator Pattern)
DanteServer ──────> ResponseObserver (Observer Pattern - Subject)

TermComponent <──── TreeIterator (Visitor-like con Iterator)

ClientResponseHandler ──────> ResponseObserver (Observer Pattern)

RequestInvoker ──────> Command (Command Pattern - Invoker)
```

Sequenza di Interazione

Fase 1: Client Pre-elabora il Testo

```
DanteClient
    │
    ├─> tokenize(text) → List<Term>
    │
    ├─> buildSemanticTree(terms) → TermComponent (Composite)
    │
    └─> compressTree(tree) → byte[] (Strategy)
```

Fase 2: Invio Asincrono

```
DanteClient
    │
    └─> ProcessRequestCommand.execute() (Command)
            │
            └─> RequestInvoker.executeAsync()
                    │
                    └─> DanteServer.receiveRequest(data)
```

Fase 3: Server Interpreta

```
DanteServer
    |
    ├─> decompressData(data) → TermComponent (Strategy)
    |
    ├─> interpretTree(tree) (Iterator - scorrimento non standard)
    |       |
    |       └─> NonStandardTreeIterator.traverse(root)
    |               |
    |               └─> visitCompositeTerm() / visitTerm()
    |
    └─> sendResponse(response)
```

Fase 4: Notifica Asincrona

```
DanteServer
    |
    └─> notify() (Observer)
            |
            └─> ClientResponseHandler.update(response)
                    |
                    └─> DanteClient riceve la risposta
```

Giustificazione dei Pattern

1. Composite Pattern

Motivazione: Il client deve rappresentare la struttura semantica del testo come albero gerarchico di termini, dove alcuni nodi sono termini semplici (foglie) e altri sono composizioni di sottostrutture (nodi compositi).

Applicazione:

- `TermComponent` : interfaccia comune
- `Term` : foglia (termine singolo)
- `CompositeTerm` : nodo composito (struttura complessa)

2. Strategy Pattern

Motivazione: L'algoritmo di compressione è "proprietario" e deve essere intercambiabile senza modificare il client o il server.

Applicazione:

- `CompressionStrategy` : interfaccia strategia

- `ProprietaryCompression` : algoritmo concreto
- Client e Server mantengono un riferimento alla strategia

3. Iterator Pattern

Motivazione: Il server deve interpretare l'albero usando uno scorrimento "non standard", separando l'algoritmo di attraversamento dalla struttura.

Applicazione:

- `TreeIterator` : interfaccia iteratore
- `NonStandardTreeIterator` : implementazione concreta
- `TermComponent.accept()` : permette al visitatore di operare sui nodi

4. Command Pattern

Motivazione: Le richieste client→server sono asincrone e devono essere incapsulate, accodate ed eseguite indipendentemente.

Applicazione:

- `Command` : interfaccia comando
- `ProcessRequestCommand` : comando concreto che incapsula la richiesta
- `RequestInvoker` : gestisce l'esecuzione asincrona
- `DanteServer` : receiver che esegue l'operazione

5. Observer Pattern

Motivazione: Client e server comunicano in modo asincrono. Il client deve essere notificato quando il server ha completato l'elaborazione.

Applicazione:

- `ResponseObserver` : interfaccia observer
- `ClientResponseHandler` : observer concreto
- `DanteServer` : subject che notifica quando la risposta è pronta
- `DanteClient` : può registrarsi come observer sul server

Note di Implementazione

1. **Separazione delle responsabilità**: Ogni pattern risolve un aspetto specifico del sistema:
   - Composite → Struttura dati
   - Strategy → Algoritmo variabile

- Iterator → Attraversamento personalizzato
- Command → Gestione richieste asincrone
- Observer → Notifiche asincrone

2. **Comunicazione asincrona**: Il Command pattern gestisce l'accodamento e l'esecuzione asincrona, mentre l'Observer pattern gestisce la notifica del completamento.
3. **Estensibilità**:
   - Nuovi algoritmi di compressione → nuove ConcreteStrategy
   - Nuove modalità di attraversamento → nuovi ConcreteIterator
   - Nuovi tipi di richieste → nuovi ConcreteCommand
4. **Accoppiamento lasco**: Client e Server sono disaccoppiati grazie a:
   - Command (richieste incapsulate)
   - Observer (notifiche indirette)
   - Strategy (algoritmi intercambiabili)