UNIVERSITÀ DEGLI STUDI DI FIRENZE

# PROGETTO METODOLOGIE DI PROGRAMMAZIONE


CHRISTIAN TANCI
MATRICOLA
7050561

ANNO ACCADEMICO 2023/2024

# INTRODUZIONE AL PROGETTO

Il progetto vede la realizzazione di una rete con topologia ad albero.
I nodi che compongono la rete sono identificati da un indirizzo ip, e organizzati in una gerarchia.

I nodi sono distinti due tipologie principali: router ed endpoint.

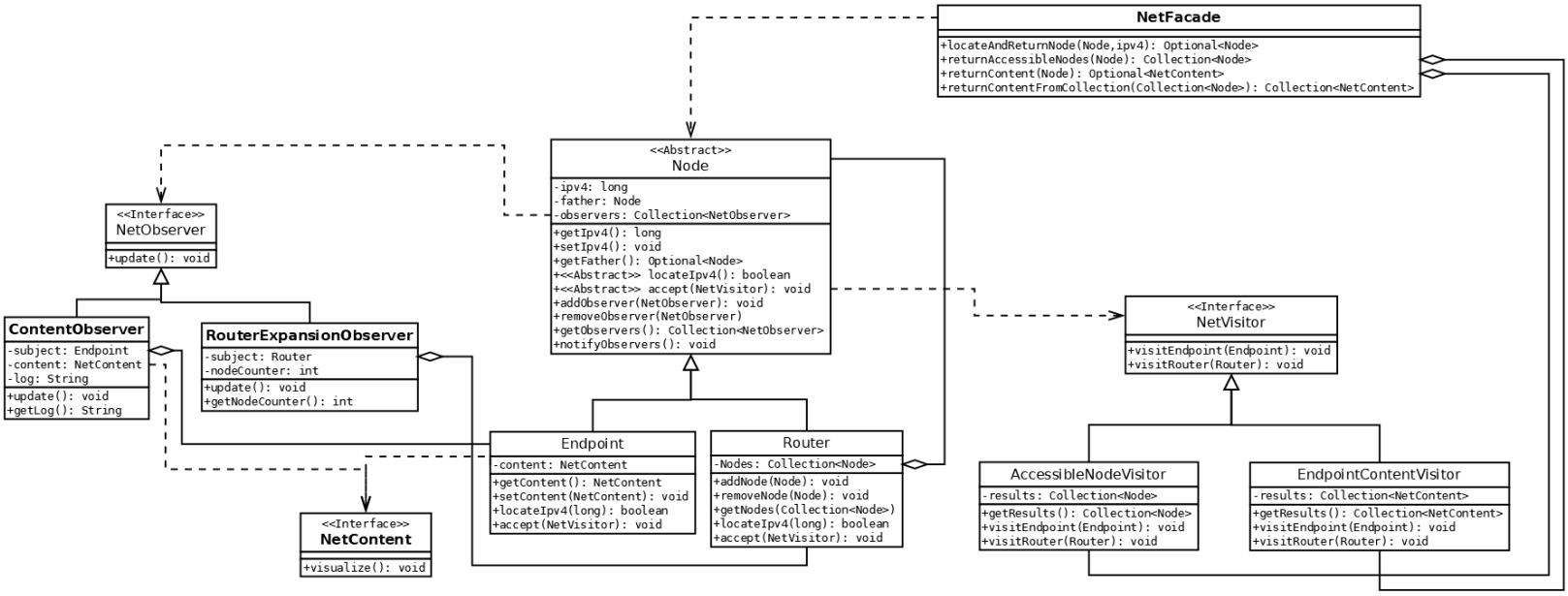
- Il router è l'entità che si occupa di mantenere l'insieme di nodi a cui è collegato. Può essere visto a livello pratico come un router, uno switch o qualsiasi altro elemento di rete pensato per instradare pacchetti.
  A livello di codice ciò è realizzato tramite una classe che mantiene una collezione dei nodi a cui è collegato.
- L'endpoint è invece pensato per essere un nodo conclusivo, a cui è possibile arrivare ma non proseguire. Può essere visto come un calcolatore, un server, una stampante.
  Qualsiasi cosa che possa contenere uno stato o un motivo per essere usato.

Questi sono i concetti su cui si fonda la struttura del progetto.
Tutte le classi realizzate e i pattern utilizzati hanno il fine di fornire operazioni da eseguire sui nodi della rete per gli utenti, cercando di mantenere il codice aperto ad estensioni e permettendo un'ampia riusabilità grazie all'uso di astrazioni.

## PATTERN UTILIZZATI E UML

Di seguito la lista di pattern utilizzati:

- **composite**, usato per realizzare la struttura ad albero della rete.
- **visitor**, usato per aggiungere operazioni alla struttura.
- **observer**, usato per aggiungere funzionalità ai nodi mantenendo alto il disaccoppiamento.
- **facade**, usato per fornire un'interfaccia più di alto livello per gli utenti del programma.



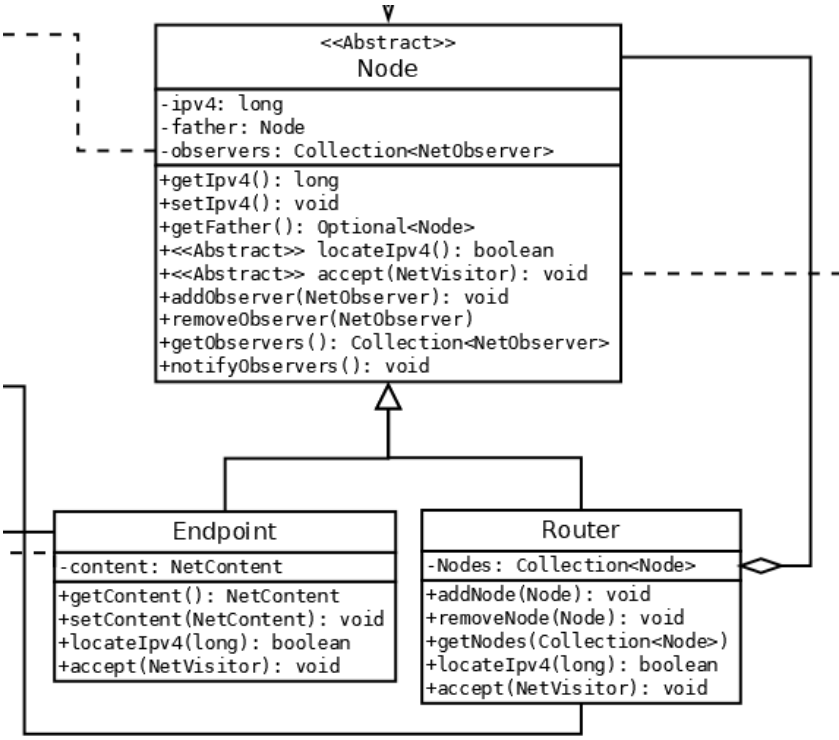

Come è visibile anche dal grafico UML, sono qua riportate le relazione tra le classi del programma:

- Le classi Endpoint e Router estendono la classe Node.
- Le classe ContentObserver e RouterExpansionObserver implementano NetObserver.
- Le classi AccessibleEndpointsVisitor e EndpointContentVisitor implementano NetVisitor.
- La classe Endpoint mantiene e opera su oggetti che estendono NetContent.
- La classe router mantiene e opera su oggetti di classe node.
- La classe Node mantiene oggetti che estendono NetObserver e NetVisitor.
- Le classi ContentObserver e RouterExpansionObserver mantengono e operano rispettivamnte con oggetti di classe Endpoint e Router.
- La classe NetFacade mantiene e opera su oggetti di classe AccessibleEndpointsVisitor e EndpointContentVisitor, e opera con oggetti di classe Node.

Il grafico UML sarà inoltre visibile durante la discussione di ogni pattern.

## PATTERN COMPOSITE

Il pattern composite è stato utilizzato per realizzare la struttura ricorsiva della rete, mantenendo distinte le due tipologie di nodi ma permettendo di trattarle in modo uniforme.




Il pattern è composto da tre classi:

- **Node**, che rappresenta l'astrazione comune(ovvero component);
- **Router**, che memorizza componeneti figli e possiede operazioni relative alla loro gestione(ovvero Composite);
- **Endpoint**, il componente foglia che non può possedere figli(ovvero leaf).

**NODE**

La classe astratta node è sicuramente la classe più vasta all'interno del programma, dato il ruolo centrale che ricopre.
Essa infatti è una classe che non solo contiene metodi ed attributi relativi al pattern composite, ma anche del pattern observer e visitor.
Gli aspetti di cui discuteremo adesso saranno però relativi esclusivamente al pattern composite.

Un nodo è identificato dal suo indirizzo ip, che all'interno di Node è rappresentato dal campo ipv4.
Questo campo è ciò su cui si basa l'operazione principale del composite.
Infatti per essere applicato correttamente il pattern necessita di un'operazione da delegare alle classi figlie; questa operazione è locateIpv4().
Essa permette di controllare partendo da un nodo se è possibile raggiungere il nodo con attributo ipv4 pari a quello passato come parametro.

Per mantenere la rete collegata è necessario inoltre avere un riferimento al nodo padre, anch'esso un'aspetto accettato dal pattern. Il riferimento è mantenuto nel campo father. Dato che è accettata una condizione dove il nodo padre non è presente, ad esempio per il nodo radice, ho deciso di gestire il campo facendo restituire al suo getter un Optional, in modo da poter usare tranquillamente oggetti null. Proprio per questo è inoltre presente un costruttore che non prende un parametro father e imposta il campo null.

Infine anche hashcode e equals sono stati ridefiniti per permettere una distinzione di oggetti Node anche sull'attributo ipv4.

**ROUTER**

La classe concreta Router rappresenta la classe composite nel pattern.
Mantiene in una collezione l'insieme di oggetti Node figli.
Possiede inoltre i metodi addNode e removeNode per la gestione dei figli. Essendo presente un riferimento all'oggetto padre, il riferimento dei figli aggiunti/rimossi è aggiornato automaticamente.

Dato lo scopo del programma ho preferito utilizzare un design per il type safety, mettendo quindi le operazioni di gestione non nella classe astratta ma bensi nella classe composta.

In questo modo non è possibile richiamare quelle operazioni su oggetti Leaf, evitando cosi di rischiare errori a runtime e non lasciando metodi con implementazioni vuote ma sacrificando uniformità.

Implementa l'operazione locateIpv4 controllande se il proprio ipv4 corrisponde a quello cercato, e in caso negativo inoltra l'operazione ai suoi componenti figli.

**ENDPOINT**
Infine la classe concreta Endpoint equivale alla classe leaf del pattern.
A differenza della superclasse mantiene un campo content di tipo NetContent che sarà approfondito successivamente.

Ero indeciso se rendere NodeContents un'interfaccia o una classe astratta:

- Ciò che richiedevo a oggetti che avrebbero esteso NodeContens era semplicemente un operazione, senza

**NETCONTENT**
Il tipo NetContent rappresenta il contenuto che ogni endpoint dovrebbe mantenere per essere un nodo rilevante all'interno della rete. Lo scopo del progetto è infatti quello di creare l'infrastruttura della rete, offrire  operazioni per modificarla e operazioni per accedere ai dati che contiene.
Non è quindi presente assolutamente nessuna implementazione concreta di questi dati(a parte come classi mock).
L'astrazione NetContent è presente per rappresentare oggetti che estendendo l'interfaccia andanno a ricoprire quel ruolo.
Il cosa può ricoprire il ruolo di contenuto è in realtà molto vasto, nella mia classe mock ho semplicemente rappresentato un utente che manteva un nome e dei dati sottoforma di stringa ma un uso molto probabile sarebbe di mantenerci un riferimento ad un oggetto composto con a sua volta una sua logica.
All'interno dell'interfaccia è comunque presente un metodo astratto visualize. Esso è pensato principalmente per dare una rappresentazione dell'oggetto. Non avendo veramente fatto uso nel programma del metodo esso potrebbe risultare inutile, ma lo scopo del programma era appunto quello di creare l'infrastruttura della rete e quindi esso era pensato per essere lasciato non ben definito.

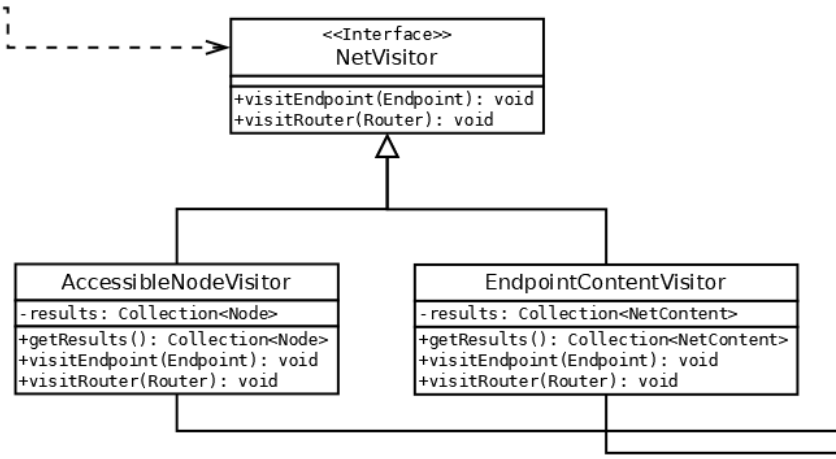
Come alternativo avevo considerato all'inizio di fare uso della classe Object invece di creare una mia interfaccia.
La classe Object avrebbe infatti permesso il massimo grado di estendibilità da parte degli utenti o di altri programmatori, lasciando libera scelta sul tipo di oggetti da usare.
Ciò che mi ha infine fatto decidere per l'interfaccia è stato che usare la classe Object avrebbe portato ad un uso troppo vario di oggetti, che sarebbe stato si permissivo ma anche incontrollato, almeno a livello di compilazione.

Alla fine ho deciso di realizzare il tutto con un interfaccia, ma questo è sicuramente l'aspetto di cui sono meno sicuro, probabilmente esiste un modo migliore per gestire la situazione.

# PATTERN VISITOR

Il pattern visitor è stato utilizzato per aggiungere operazioni alla struttura realizzata dal composite, senza appesantire inutilmente le classi che lo compongono.
Le classi che implementano il pattern sono le seguenti:

- **NetVisitor**, l'astrazione che fornisce le operazioni comuni da implementare.
- **AccessibleEndpointsVisitor**, classe concreta che permette di ottenere una collezione di endpoint raggiungibili da un certo nodo.
- **EndpointContentVisitor**, classe concreta che fornisce un metodo di accesso a oggetti NetContent.




**NETVISITOR**

L'interfaccia NetVisitor stabilisce i metodi astratti che i visitor concreti dovranno implementare.
Dato che le classi concrete della struttura da visitare sono due avremo rispettivamente visitEndpoint e visitRouter.
Nella classe astratta comune Node è stata aggiunta l'operazione accept, che prende come parametro un visitor. Essa è poi implementata dalle sottoclassi.

**ACCESSIBLEENDPOINTSVISITOR**

Come abbiamo gia detto, la rete ha una struttura gerarchica e da un nodo si accede a quelli di livello inferiore.
Questa classe permette di restituire l'insieme di oggetti Endpoint raggiungibili che sono mantenuti nel campo results, una collezione di nodi.

Le operazioni astratte ereditate da NetVisitor sono state implementate nel seguente modo:

- visitEndpoint aggiunge il suo riferimento alla collezione, per indicare che è raggiungibile dal nodo su cui è stato chiamato il visitor.
- visitRouter non aggiunge se stesso alla collezione, ma bensi inoltra la chiamata ai suoi oggetti figli.

Ho fatto uso della classe astratta Node nonostante per struttura dell'operazione tutti gli oggetti contenuti saranno oggetti Endpoint per non fare uso di classi concrete staticamente ed evitare quindi dipendenze.

Ogni istanza di questa classe è pensata per essere monouso dato che non esiste un metodo setter per ripristinare results.

**ENDPOINTCONTENTVISITOR**
Il tipo Endpoint contiene un il campo content, ma la classe astratta Node non possiede metodi per accedere ad esso senza uso di downcast.

La classe EndpointContentVisitor permette di risolvere il problema, implementando come segue le operazione di NetVisitor:

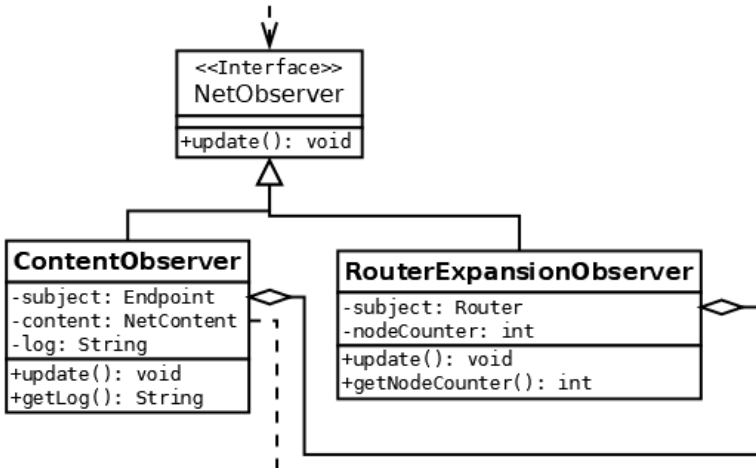
- visitEndpoint aggiunge il suo campo NetContent alla collezione result, mentre visitRouter non svolge alcuna operazione, neanche di inoltro.

Nonostante questo il campo result rimane una collezione come il precedente visitor. Questo perchè nonostante la singola chiamata su nodo potrà al massimo aggiungere un elemento alla collezione, nulla vieta di utilizzare sequenzialmente il visitor su una collezione di nodi, in modo da non dover istanziare un nuovo visitor ogni volta che vogliamo accedere ad un contenuto di un endpoint.
Un esempio di esso sarà svolto all'interno di NetFacade.

## PATTERN OBSERVER

Il pattern observer è stato utilizzato per permettere un controllare determinate caratteristiche dei nodi componenti la rete.
Le classi che implementano il pattern sono le seguenti:

- **NetObserver**, l'astrazione comune del pattern.
- **ContentObserver**, un observer che si aggiorna quando il contenuto di un endpoint viene modificato.
- **TrafficObserver**, un observer che mantiene nel suo stato l'attuale grandezza di nodi a cui è collegato un router.




Il ruolo di subject è ricoperto dalla classe Node, e di conseguenza dalle classi concrete che la implementano.
Gli observer che ho realizzato hanno infatti un riferimento concreto alle classi che implementano Node, e sono quindi pensati per essere usati con

### CONTENTOBSERVER

La classe ContentObserver mantiene multipli campi che permettono di svolgere il suo lavoro:

- subject mantiene il riferimento fisico all'endpoint osservato dall'observer;
- content mantiene l'ultimo NetContent registrato dall'observer;
- infine log mantiene appunto un log delle operazioni che sono state svolte su quello stato.

Quando viene chiamato un setContent sull'endpoint viene lanciata una notifyAllObservers; l'observer decide se è necessario aggiornare il suo stato e il log tramite un confronto.
L'esistenza di un observer che esegue log delle modifiche avvenute nel contenuto di un endpoint potrebbe ad esempio servire per monitorare gli accessi.

Inizialmente all'interno del log doveva essere presente un mesaggio completo di data e minutaggio della modifica al contenuto tramite la classe LocalDateTime.
Il testing è però risultato molto arduo, dato che il tempo indicato all'interno della stringa e quello nel test alcune volte non combaciavano.
Per evitare errori imprevisti ho quindi dovuto passare ad una soluzione più semplice e che non fa uso di istanti di tempo.

### TRAFFICOBSERVER

La classe TrafficObserver permette di controllare l'espansione del primo livello della sottorete di un router. Controllare la dimensione potrebbe rivelarsi utile ad esempio in situazioni di sovraffaticamento della rete; ad esempio si potrebbe mantenere un insieme di TrafficObserver che controllano dei router e in caso un router inizi a possedere una quantità non sicura di figli spostare alcuni nodi.

L'implementazione di ciò è molto basilare: quando un router aggiunge o rimuove un nodo dalla lista dei suoi figli esso svolge una notifyObservers.
Quindi il TrafficObserver controlla se il numero di nodi è cambiato e aggiorna il suo contatore.

## PATTERN FACADE

Il facade che ho realizzato offre metodi agli utenti del sistema che permettono di interagire con la rete in modo più semplice.
In più esso nasconde i dettagli di implementazione del visitor e del composite agli utenti, che non dovrebbero essere obbligati a conoscere.

- ***locateAndReturnNode*** ricerca all'interno della rete il nodo con l'ipv4 richiesto.
  Esso prima richiama locateIpv4 per controllare che il nodo sia effettivamente nella rete, per poi fare uso del visitor.
  In questo modo si evita di dover ricercare all'interno di una collezione che contiene tutti gli endpoint di una rete un endpoint che neanche esiste.
  Come tipo di ritorno del metodo ho fatto uso di Optional<Node>, in modo da evitare di restituire null in caso il nodo non esista.

- **returnAccessibleEndpoint** permette di ottenere gli stessi risultati di utilizzare un AccessibleEndpointsVisitor su un nodo, ma nascondendo i dettagli di implementazione del visitor all'utente.
  Restituendo una collezione, non è necessario fare uso di Optional. Infatti nel caso non siano presenti alcuni endpoint si avrà semplicemente una collezione vuota.

- **returnContent** permette di ottenere il contenuto di un endpoint. Come parametro il metodo prende però un Node: questo perché se si avesse un oggetto Endpoint a livello statico sarebbe possibile semplicemente richiamare .getContent().
  Il fine di questo metodo(come del visitor stesso) è quello di poter ottenere gli stessi risultati di getContent() sulla classe astratta.
  Ricordo che per struttura il visitor EndpointContentVisitor non svolge alcuna operazione in caso sia richiamato visitRouter().
  Per questo il tipo di ritorno usato è Optional<NetContent>, sempre per evitare di restituire oggetti potenzialmente null.
  In più all'interno dello stream è stato fatto uso di un filter che rimuove i NetContent null, questo perché findFirst() lancia una NullPointerException in caso l'elemento restituito sia null.
  Invece di gestire l'eccezione ho preferito far si che il metodo restituisca un empty Optional anche in caso il NetContent sia null.

- **returnContentFromCollection** è un metodo con comportamento simile a returnContent ma pensata per prendere come paramentro una collezione di nodi.
  Similmente a getContent si otterrà i contenuti di tutti Endpoint.
  Il parametro di ritorno è infatti Collection<NetContent>. Anche qua non è necessario fare uso di Optional, restituendo al massimo una collezione vuota.

## TEST

I test sono stati realizzati usando JUnit 4.13.0.
Inolte ho fatto uso della libreria esterna AssertJ v.3.14.0.

Nei test ho fatto uso di elementi null per rendere meno verbosi i test ma solo a condizione che non potesse causare errori come NullPoiterException.

Per testare classi che utilizzano oggetti di tipo NetContent ho fornito un'implementazione fittizia, posizionate all'interno del package progetto.mp.util.
MockNetContent è stata realizzata usando due campi, uno per indicare un nome utente e uno per dei dati, entrambi come stringhe. Implementazione banale usata appunto solo per dei test.
L'operazione visualize è stata implementata in modo fittizio dato che all'interno dei test non se ne faceva uso.