FARE (DOMANI)

Comportamentale

Design Pattern Chain of Responsability

Intento

Permette di gestire UNA RICHIESTA ANDANDO AD INOLTRARE AL NODO SUCCESSIVO DI UNA CATENA. (FINCHÉ NON SARÀ GESTITA DA QUALCHE NODO)

- Si deve evitare di accoppiare il mandante di una richiesta con il ricevente.
- Ebbene non si conosca il mandante, la richiesta viene mandata a catena a tutti.
  - *Chi non riesce a gestire la richiesta la manda alla successiva catena (Chain)*

> *Quando si tocca un bottone (*Conferma*) esso cosa fa? Conferma i dati inseriti oppure fa anche altre cose?*

Esempio

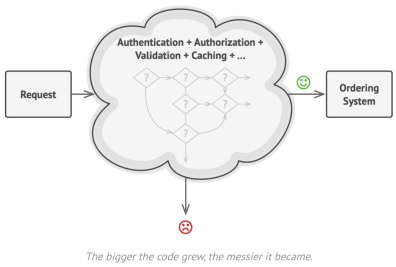
Se si ha un sistema che ha la possibilità di fornire *Aiuto* sugli elementi puntati dal mouse (*classica finestra a comparsa*). La funzionalità di Aiuto è inserita su precisi elementi non subito capibili...Se invece si usa un pulsante Aiuto generico, ci si aspetta che qualcuno risponda a tale richiesta (*può rispondere* -> o l'elemento specifico che l'utente ha selezionato col cursore oppure, se l'elemento selezionato dall'utente non ha delle informazioni di aiuto da dare, risponde chi di dovere, seguendo una *catena di richieste*)

Motivazione

Formalizzando l'esempio sopra descritto...
Consideriamo una interfaccia utente dove l'utente può richiedere aiuto su parti dell'interfaccia. Per es. un bottone può fornire informazioni di aiuto.
Se non esiste un'informazione specifica allora il sistema dovrebbe fornire il messaggio d'aiuto del contesto più vicino (ad es. la finestra)

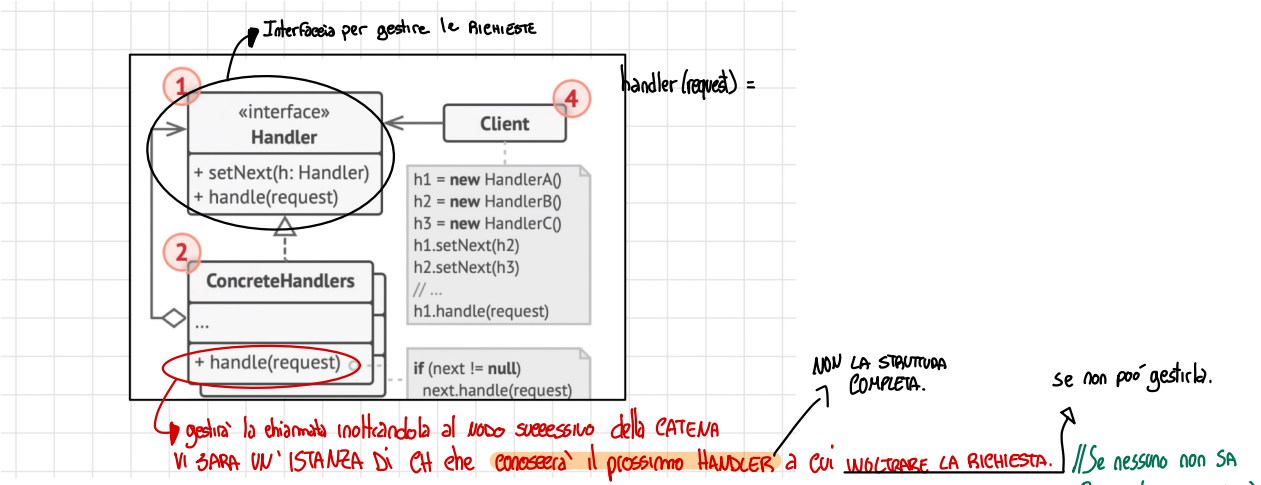
- Il problema: l'oggetto che fornirà il messaggio d'aiuto non è conosciuto dall'oggetto (es. Button) che inizia la richiesta
- Disaccoppiare mandante e ricevente

Esempio 2: autenticazione

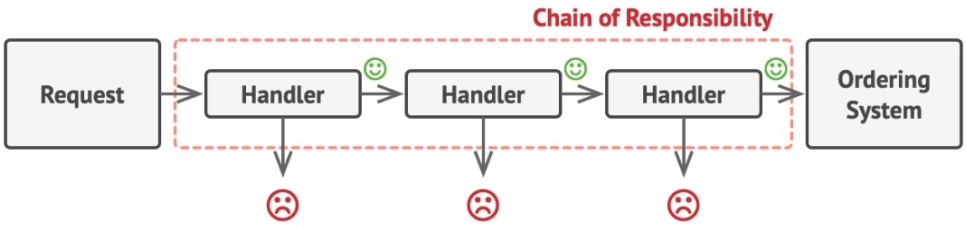


The bigger the code grew, the messier it became.

Soluzione



- *Handler* definisce **l'interfaccia** per gestire le richieste
- La `handleRequest()` in *Handler* (*H*) potrebbe essere "Mostra aiuto". Questa funzione ***gestisce una richiesta***
- Le *ConcreteHandler1 /2* (*CH*) implementeranno `handleRequest()`
- Ogni elemento conosce un altro elemento dello stesso tipo.
- A run-time ci sarà un'istanza di *CH* e quest'ultimo conoscerà un altro H (che sarà poi *CH1* o *CH2*) che rappresenterà ***il successivo elemento della catena***
- Quando *CH1* deve eseguire la `handleRequest()` si controlla se egli stesso può gestirla. Se non può gestirla la rimanda al successivo elemento della catena, richiamando `handleRequest()` proprio su *successor* (*visto che lo conosce*)
- `handleRequest()` potrebbe avere $0+$ parametri
- *Il Client non conosce quale* CH *specifico risponderà alla richiesta*.
  - Egli inizia la catena di richiesta su un *CH* iniziale specifico
  - non è detto che risponda proprio la classe sulla quale è stata chiamata la richiesta inizialmente



Handlers are lined up one by one, forming a chain.

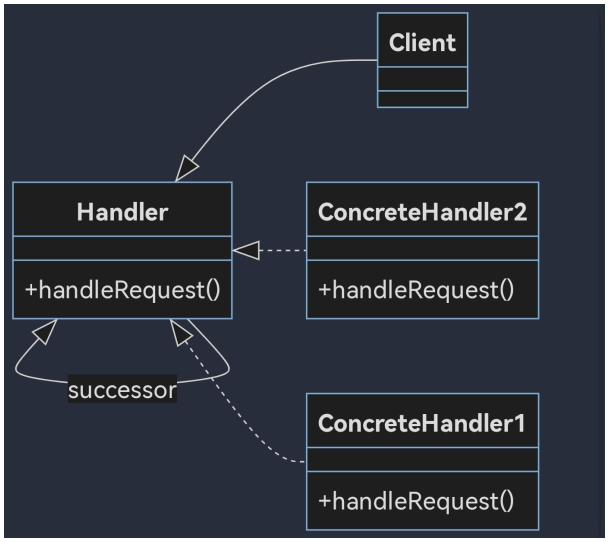
Conseguenze

- Un oggetto della catena non conosce la struttura COMPLETA della catena ma **esclusivamente il successivo elemento**
- Si possono **aggiungere/rimuovere elementi** dalla catena per aumentare/diminuire la propagazione della la richiesta in questione manipolando la *lista concatenata*
- La propagazione della **richiesta può arrivare alla fine della catena**. In questo caso nessun elemento sa rispondere e quindi la richiesta potrebbe non essere **MAI soddisfatta**

Implementazione

- L'operazione `handleRequest()` definita in H potrebbe essere un'operazione non solo definita ma anche un richiamo all'elemento successivo.
- *H* potrebbe anche non essere un'interfaccia (ma generalmente è meglio che lo sia) proprio per casi come questi
- Questo Pattern può essere **implementato in una gerarchia già esistente**, come nel *Composite*: Impone ai vari *ConcreteComponent* di implementare un'operazione di risoluzione/propagazione di una richiesta
- Alcune classi facenti parte di Composite, potrebbero implementare diversi Design Pattern in base al proprio ruolo (chiaramente questo è valido per applicazioni non banali)
- Ogni *CH* segue una **CONVENZIONE** sui dati che determina il tipo di parametro (se definito) passato alla richiesta in questione. Quindi ogni CH devono sapere come interpretare il parametro passato a `handleRequest()` (quindi ci deve essere **coerenza**)

Diagramma delle classi Mermaid



Design Patter Prototype => Creazionale

Consente di creare nuovi oggetti copiandone uno già esistente. (Utile quando la CREAZIONE di oggetti è troppo DISPENDIOSA)

Intento

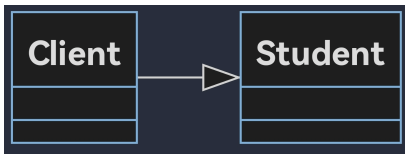
- E' un design pattern **CREAZIONALE**
- Quando si creano le istanze, se ne usano altre che fa da base per la creazione di altre istanze. Piuttosto che specificare la classe di cui avere l'istanza, si usa un'istanza già presente, quindi iniziale, che fa proprio da ***PROTOTIPO*** da cui copiare. La nuova istanza si porta i dati del prototipo

Motivazioni

- SI ha un'istanza a run-time che ha vari dati.
- Lego una classe che vuole una nuova istanza (richiedente) alla classe che viene usata per creare l'istanza

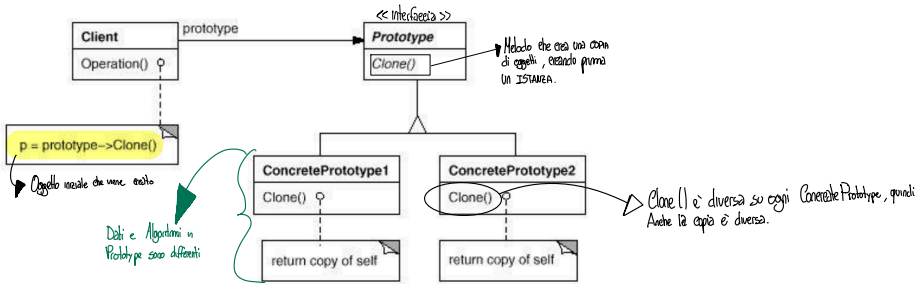
Esempio: Se devo istanziare Client, si deve istanziare Student.

```
public class Client{
        //...
        v = new Student();
}
```

In questo caso si ha: 

> *Si vuole evitare di avere questa dipendenza. Si potrebbe usare un `FactoryMethod()` ma si genererebbe una vasta gerarchia di Creator per slegare il Client dal ConcreteProduct

Soluzione



- *Prototype* (P) definisce l'operazione `clone()` che copia l'istanza
- Ogni *ConcretePrototype* (CP) implementa `clone()`
- *Client* si lega a P
- La gerarchia di *CP* è equivalente alla gerarchia dei *ConcreteProduct* (*in Factory*) e assomigliano ai "*prodotti*" che ci servono. Qui, però, serve il metodo `clone()`
- P deve includere la definizione delle operazioni per mantenere la compatibilità con i sottotipi CP. Cioè devono esistere altre operazioni (implementate nei sottotipi) che specificano proprio quel *P*
- `clone()` fa le seguenti operazioni: si **crea** un'istanza e **copio** lo stato iniziale nella nuova istanza
- **I dati e gli algoritmi implementati** in CP1 **sono diversi** rispetto a tutti gli altri CP, appunto perchè non *si fa duplicazione di codice*.
  - Ne segue che `clone(),` su ogni CP, è diversa per ogni istanza e quindi la copia è diversa e non può essere unica per tutti i CP
- Il Client conose il tipo più generale ma a run-time, **tramite clonazione,** avrà il **tipo più specifico** (*quindi senza indicare la classe esplicitamente*). Da questo punto, quindi, verrà creata l'istanza di uno specifico CP ma, comunque, il Client conosce il tipo più generico P

> *Si evita di legare chi ha bisogno della nuova istanza con la classe che ha quella particolare istanza*

Conseguenze

- Client può usare CLASSI SPECIFICHE SENZA CONOSCERLE.
- Possiamo Creare Oggetti Complessi FACILMENTE.
- Riduce il Tempo per la Creazione di oggetti

Si può AVERE una VARIANTE con il REGISTRO tiene traccia di diversi tipi di oggetti e FORNIRE LE COPIE Quando RICHIESTO.

Il Registro tiene un Associazione (chiave:Valore) e si possono FARE

Shallow Copy = Condivide gli stessi riferimenti agli oggetti della COPIA ORIGINALE.

Deep Copy = Copia in PROFONDO dei DATI. (evitando le dipendenze tra classi)

- Dà la possibilità di registrare nuovi P a runtime
- Il client può usare classi specifiche (sottoclassi) senza conoscerle direttamente
- Si può aggiungere un ***REGISTRO*** che tiene i prototipi che si possono usare: La classe generica può cercare nel registro (tramite una chiave) e istanziarne uno preciso
- Se diversi oggetti hanno uno stato diverso, rappresentano dei comportamenti diversi. Di conseguenza, pur avendo CP1 (per esempio) la si fa partire da stati diversi. Visto che lo stato cambia, allora clone() sarà diverso (anche per la stessa classe). La programmazione è semplificata per il numero di linee di codice che si scrivono.
  - Si usa (tramite registro o clonazione) da quale stato partire per avere quello stato diverso
  - Si può definire uno **stato iniziale** e conservarlo
- Se in un CP ci sono attributi (tipi riferiti ad altri oggetti) allora si ha una composizione. Un nuovo prototipo può essere usato per tere una struttura diversa di oggetti, appunto

Implementazione

Si può avere una classe apposita RegistroDiPrototipi che tiene un'associazione (chiave:valore) che permette ai client di immagazzinare e recuperare i prototipi, prima di clonarli

Quando si ricopia lo stato:

- si potrebbero avere dei **riferimenti circolari**: nello stato da copiare potrebbero essere dei riferimenti successivi fino a ritornare al punto di partenza (*loop infinito*)

La copia potrebbe essere superficiale (***shallow copy***) o profonda (***deep copy***):
(si suppone di avere una classe A con 2 attributi: `int x = 5; B v = new B();` e si vuole clonare. A ha una dipendenza con B)

- Shallow copy: L'attributo di tipo primitivo *x*, verrà ricopiato nell'istanza clone e, inoltre *v* si riferirà alla stessa identica porzione di memoria.
  - Questo effetto è indesiderato perchè si vorrebbe che il clone avesse la propria dipendenza con B (separata dalla precedente). In questo caso serve la *Deep copy*, quindi bisogna implementare manualmente la clonazione.
- Deep copy: copia in modo profondo i dati evitando dipendenze "fra classi"