Il pattern Flyweight

a cura di **Angelo Furfaro**
da “Design Patterns”, Gamma et al.
e da refactoring.guru/design-patterns

Dipartimento di
Ingegneria Informatica, Elettronica, Modellistica e Sistemistica
Università della Calabria, 87036 Rende(CS) - Italy
Email: a.furfaro@unical.it
Web: http://angelo.furfaro.dimes.unical.it

Flyweight

Classificazione

- Scopo: strutturale
- Raggio d'azione: oggetti

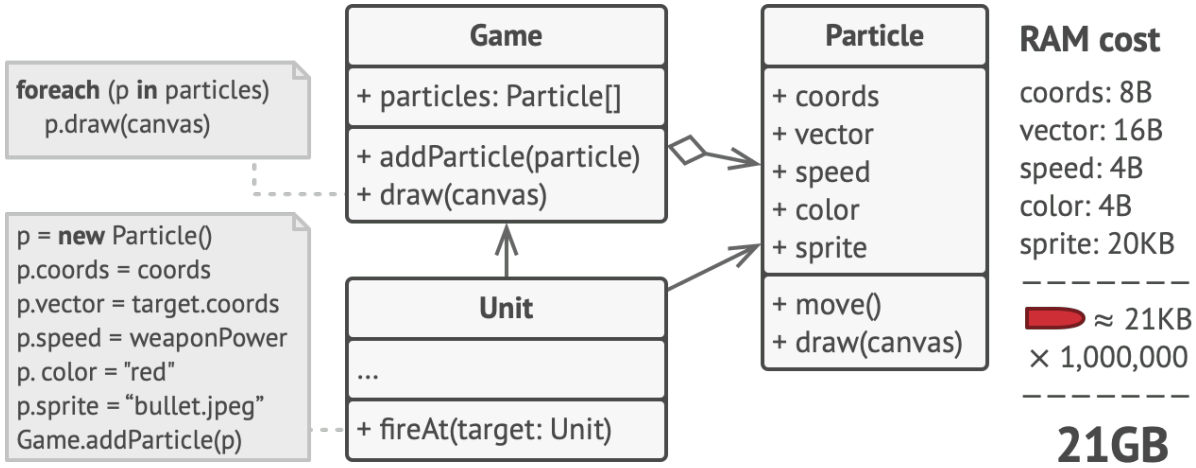
Altri nomi

Cache

Scopo

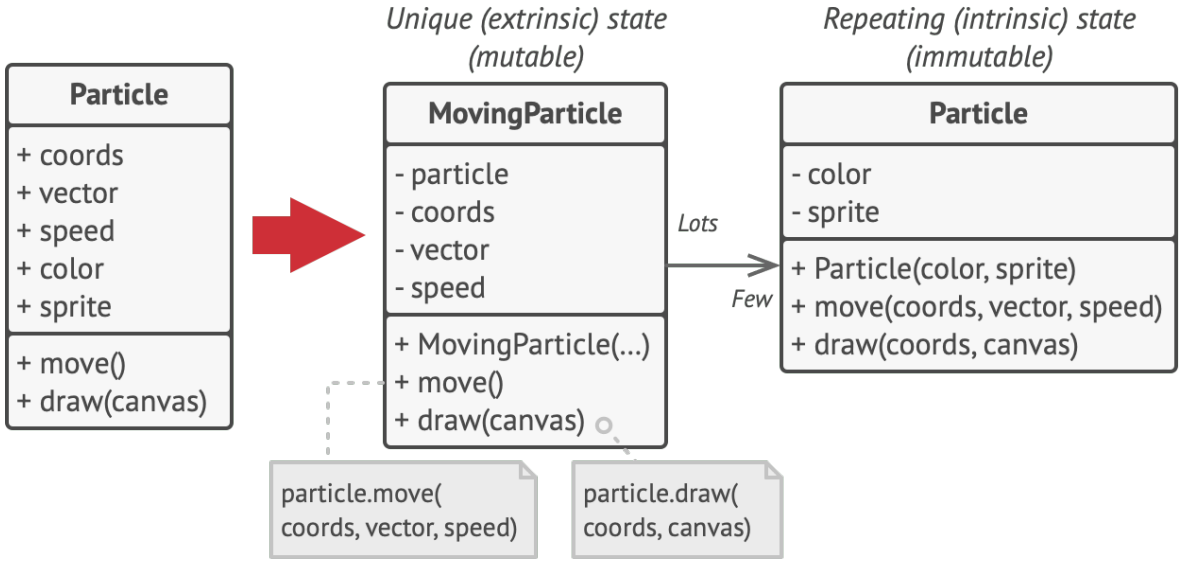
- Utilizzare la condivisione per supportare in modo efficiente un gran numero di oggetti a granularità fine.

Motivazione



- Alcune applicazioni potrebbero trarre beneficio dall'approccio a oggetti fin dalla progettazione, ma un'implementazione ingenua può risultare assai onerosa.
- Si consideri ad esempio un'applicazione consistente in un semplice videogioco in cui i giocatori si muovono su una mappa e si sparano a vicenda. Si è scelto di implementare un sistema particellare realistico e renderlo una caratteristica distintiva del gioco. Grandi quantità di proiettili, missili e schegge di esplosioni dovrebbero volare su tutta la mappa.
- Ogni particella, come un proiettile, un missile o una scheggia, è rappresentato da un oggetto separato contenente molti dati.
- Durante l'esecuzione, ad un certo punto, non c'è più spazio a sufficienza nella RAM per le particelle appena create, quindi il programma va in crash.
- Lo svantaggio di un progetto di questo tipo è il suo costo, poiché anche scenari di dimensione modesta potrebbero richiedere centinaia di migliaia di oggetti particella, che utilizzano una quantità notevole di memoria e possono introdurre un carico inaccettabile le durante l'esecuzione.
- Il pattern `Flyweight` descrive come condividere oggetti in modo da consentire il loro uso a granularità fine senza avere costi proibitivi.

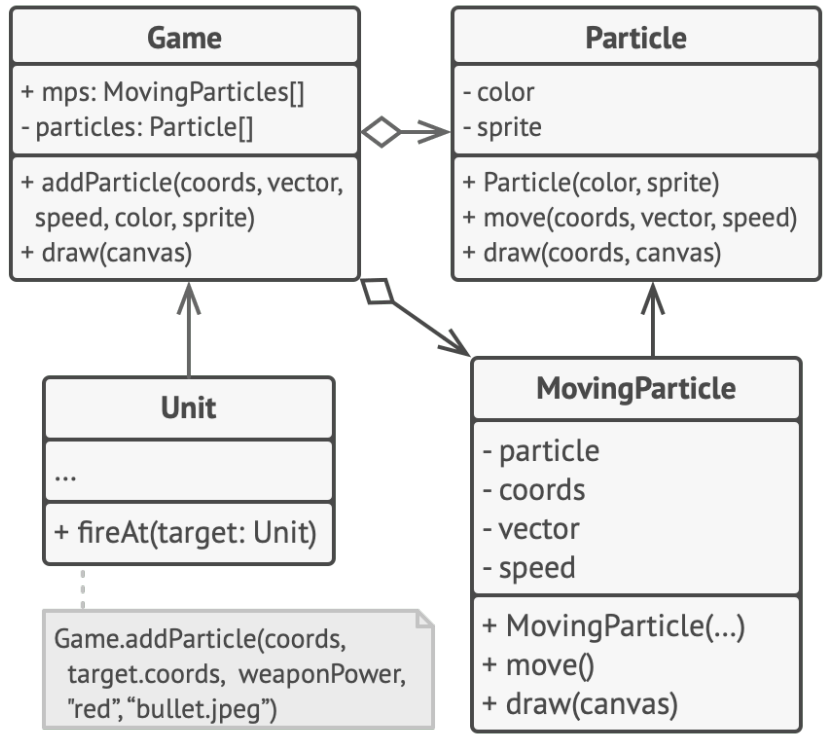
Motivazione

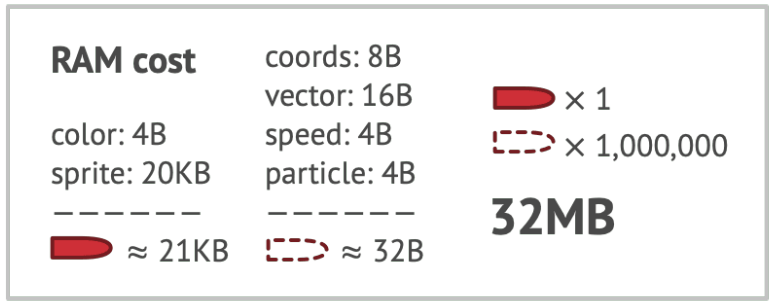


- Un *flyweight* è un oggetto condiviso che può essere utilizzato simultaneamente in più contesti.
- Gli oggetti flyweight non possono quindi fare alcuna ipotesi sul contesto in cui operano ed è pertanto fondamentale riuscire a distinguere fra stato **interno** (o *intrinseco*) ed **esterno** (o *estrinseco*) di un oggetto flyweight.
- Lo stato interno, memorizzato nel flyweight, è costituito da informazioni di stato indipendenti dal contesto in cui si trova il flyweight e pertanto può essere condiviso.
- Lo stato esterno dipende invece dal contesto in cui il flyweight opera e pertanto non può essere condiviso.
- Gli oggetti client sono responsabili del passaggio dello stato esterno al flyweight quando necessario.
- Nell'esempio i campi `color` e `sprite` consumano molta più memoria rispetto agli altri. Inoltre essi memorizzano informazioni pressoché identiche tra le particelle. Ad esempio tutti i *proiettili* avranno lo stesso colore e lo stesso *sprite*.

Motivazione

- Gli altri campi, quali le coordinate, il vettore direzione e la velocità hanno valori distinti per ogni particella e inoltre cambiano con il tempo.
- Colore e sprite corrispondono allo stato *intrinseco* mentre gli altri campi allo stato *estrinseco*.
- La classe `Particle` modella lo stato intrinseco (immutabile), la classe `MovingParticle` quello estrinseco.
- Solo tre oggetti diversi saranno sufficienti a rappresentare lo stato esterno di tutte le particelle del gioco: uno per i proiettili, uno per il missili e uno per le schegge.
- Nell'esempio lo stato intrinseco è memorizzato nell'array `particle` della classe `Game` mentre quello estrinseco nell'array `mps`.
- Una soluzione più elegante consiste nell'introdurre una classe di contesto che memorizza lo stato estrinseco e un riferimento all'oggetto flyweight che corrisponde a quello intrinseco.



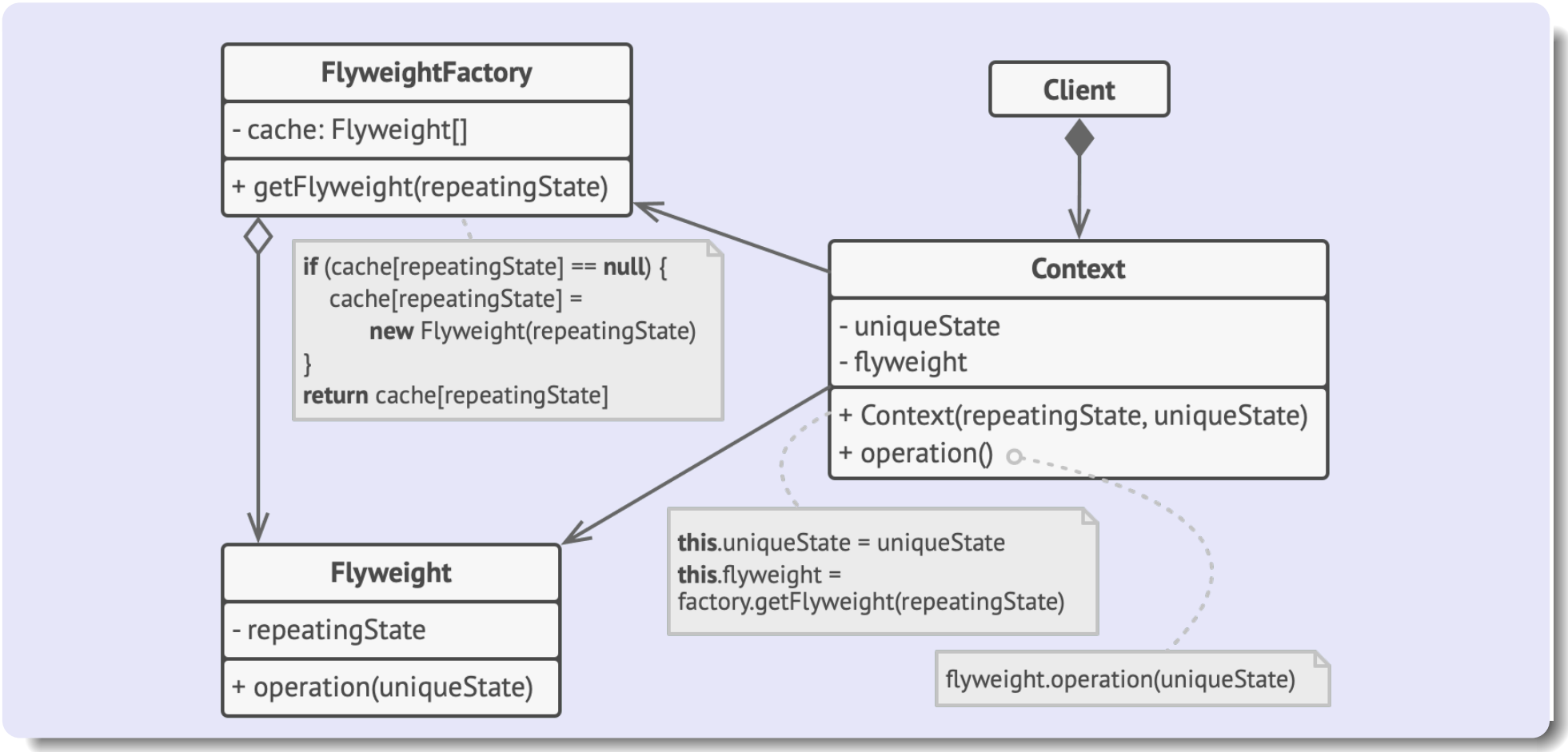


Applicabilità

L'efficacia del pattern `Flyweight` dipende molto dal modo in cui viene applicato. In particolare, il pattern `Flyweight` può essere utilizzato quando sono vere tutte le condizioni seguenti:

- un'applicazione utilizza un gran numero di oggetti;
- i costi di memorizzazione sono alti a causa dell'enorme quantità di oggetti;
- molti gruppi di oggetti possono essere sostituiti da un numero relativamente basso di oggetti condivisi, una volta rimosso lo stato esterno;
- l'applicazione non dipende dall'identità degli oggetti. Poiché gli oggetti flyweight possono essere condivisi, un controllo sull'identità restituirebbe vero per oggetti concettualmente distinti.

Struttura


FlyweightFactory
- cache: Flyweight[]
+ getFlyweight(repeatingState)
Client
Context
- uniqueState
- flyweight
+ Context(repeatingState, uniqueState)
+ operation()
if (cache[repeatingState] == null) {
cache[repeatingState] =
new Flyweight(repeatingState)
}
return cache[repeatingState]
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
flyweight.operation(uniqueState)
Flyweight
- repeatingState
+ operation(uniqueState)

Partecipanti

- **Flyweight**: dichiara un'interfaccia attraverso la quale gli oggetti flyweight possono ricevere lo stato esterno e agire di conseguenza.
- **FlyweightFactory**: crea e gestisce gli oggetti flyweight. Si assicura che i flyweight siano condivisi in modo appropriato. Quando un client richiede un flyweight, l'oggetto `FlyweightFactory` restituisce un'istanza esistente, oppure, se non esiste alcuna istanza, prima la crea e poi la restituisce.
- **Context**: contiene lo stato estrinseco, unico tra tutti gli oggetti originali. Quando un oggetto context è accoppiato con uno degli oggetti flyweight, rappresenta l'intero stato di un oggetto originale.
- **Client**: calcola oppure memorizza lo stato estrinseco dei flyweight. Dal punto di vista del client, un flyweight è un oggetto template che può essere configurato a tempo di esecuzione fornendogli dati contestuali come parametri dei suoi metodi.

Conseguenze

- L'utilizzo dei flyweight può introdurre costi aggiuntivi durante l'esecuzione dovuti all'individuazione, all'elaborazione e/o al trasferimento dello stato esterno da parte del client.
- Questi costi aggiuntivi sono compensati dal risparmio in termini di spazio di memorizzazione, che aumenta quanto più cresce il grado di condivisione tra i flyweight.
- La riduzione dello spazio totale necessario per la memorizzazione delle informazioni dipende da svariati fattori:
  - la riduzione nel numero complessivo di istanze dovuta alla condivisione;
  - la quantità di spazio necessaria per memorizzare in ogni oggetto flyweight lo stato interno;
  - la possibilità di calcolare lo stato esterno, invece della sua memorizzazione.
- Lo spazio di memorizzazione può quindi essere ridotto in due modi: attraverso la condivisione, che riduce il costo di memorizzazione dello stato interno e attraverso il calcolo dello stato esterno in luogo della sua memorizzazione, al costo di un incremento nel tempo di computazione.

Implementazione

Nell'implementazione del pattern `Flyweight` è opportuno considerare i seguenti aspetti:

- *Rimozione dello stato esterno*. L'applicabilità del pattern è largamente determinata, dalla facilità con cui si riesce a identificare lo stato esterno e a rimuoverlo dagli oggetti che si vogliono condividere. Nel caso in cui vengano individuate tante tipologie di stato esterno quanti sono gli oggetti da condividere, la rimozione dallo stato esterno non aiuterebbe a ridurre i costi di memorizzazione. Idealmente, lo stato esterno potrebbe essere completamente calcolato partendo da una struttura a oggetti separata e costruita in modo tale da ridurre significativamente i requisiti di memorizzazione.
- *Gestione degli oggetti condivisi*. Poiché gli oggetti sono condivisi, i client non dovrebbero poterli istanziare direttamente. La classe `FlyweightFactory` consente ai cient di individuare ed eventualmente di creare un oggetto flyweight particolare e di ottenere un riferimento ad esso. Gli oggetti `FlyweightFactory` spesso utilizzano una memoria associativa. La condivisione può richiedere anche una qualche forma di conteggio dei riferimenti o di *garbage collection* per reclamare la memoria occupata da un flyweight non più utilizzato. Ciò non è strettamente necessario qualora il numero di flyweight sia piccolo e noto a priori.