Un Framework per la Meta-programmazione in *Minecraft*

Libreria OOPACK

Nanni Alessandro

Alma Mater Studiorum Università di Bologna

16/12/2025

Contenuti

Minecraft come piattaforma di sviluppo

Minecraft non è solo un gioco, ma un'ambiente programmabile tramite *datapack* (logica) e *resourcepack* (risorse). Queste due cartelle costituiscono un *pack*.

Come linguaggio di programmazione si utilizza *mcfunction*, un Domain Specific Language (DSL) interpretato dal motore di gioco.

La tesi si propone di analizzare le problematiche del DSL e dell'ecosistema di file sottostante ai *pack*, per poi affrontarle mediante la progettazione e l'implementazione di una libreria Java dedicata alla meta-programmazione.

Limiti di *mcfunction*

- Assenza di variabili o strutture dati complesse: le operazioni matematiche possono essere eseguite solo su interi;
- Frammentazione: ogni funzione deve essere definita in un apposito file. I cicli devono essere implementati tramite ricorsione;
- Gestione matematica: difficoltà nel calcolare con precisione decimale i valori di funzioni quali seno, coseno e radice quadrata che richiedono *lookup table*;
- Boilerplate: definire un oggetto semplice richiede fino 7 file diversi.

```
1   data modify storage my_storage sqrt set value [
2     0,
3     1.0,
4     1.4142135623730951,
5     1.7320508075688772,
6     2.0,
…                                   …
102   10.0
103 ]
```

Codice 1: *Lookup table* per $\sqrt{x}$, con $0 \leq x \leq 100$.

Libreria OOPACK

Questa libreria Java (Object Oriented Pack) astrae la struttura di un *pack* come un albero di oggetti tipizzati.

1. Lo sviluppatore scrive codice ibrido tra Java e mcfunction, rendendolo in grado di sfruttare costrutti di un linguaggio di alto livello.
2. La libreria valida staticamente la struttura;
3. Il metodo `build()` genera l'intera gerarchia di file, fornendo zucchero sintattico per velocizzare la scrittura di componenti ripetitive.

Architettura del software

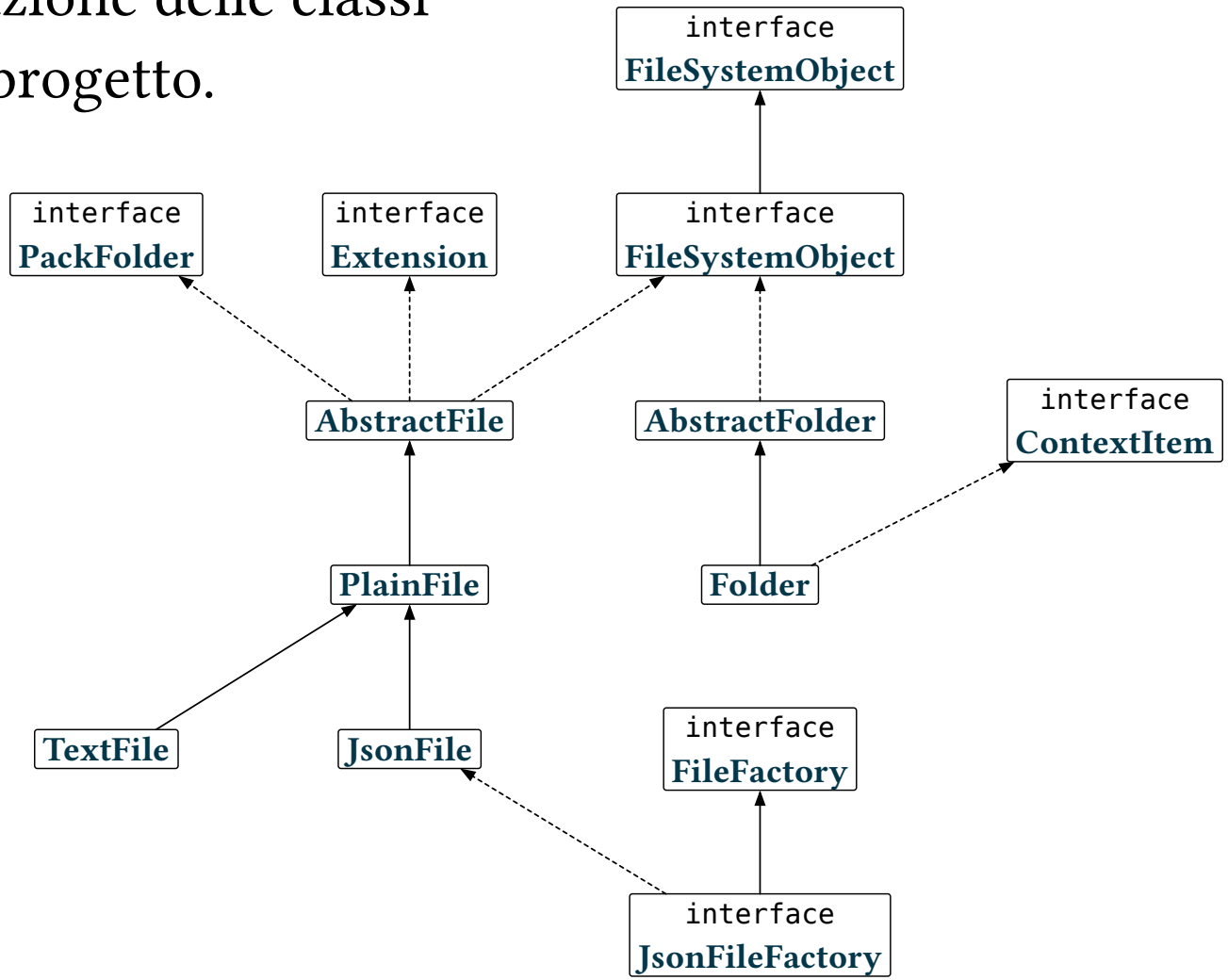
La struttura del progetto è stata raffinata iterativamente per giungere alla scrittura di un API che sia facile da leggere, utilizzare e contribuire in futuro. A tal fine sono stati impiegati numerosi *design pattern*:

Composite `AbstractFolder` sfrutta il polimorfismo per gestire i suoi contenuti, che possono essere altri `AbstractFolder` o `AbstractFile`.

Factory Ogni oggetto rappresentate un file è istanziato tramite una factory. Sono state impiegate *abstract factory* per definire le modalità con cui un oggetto può essere inizializzato.

Builder Utilizzato per la configurazione del progetto.

Rappresentazione delle classi astratte del progetto.

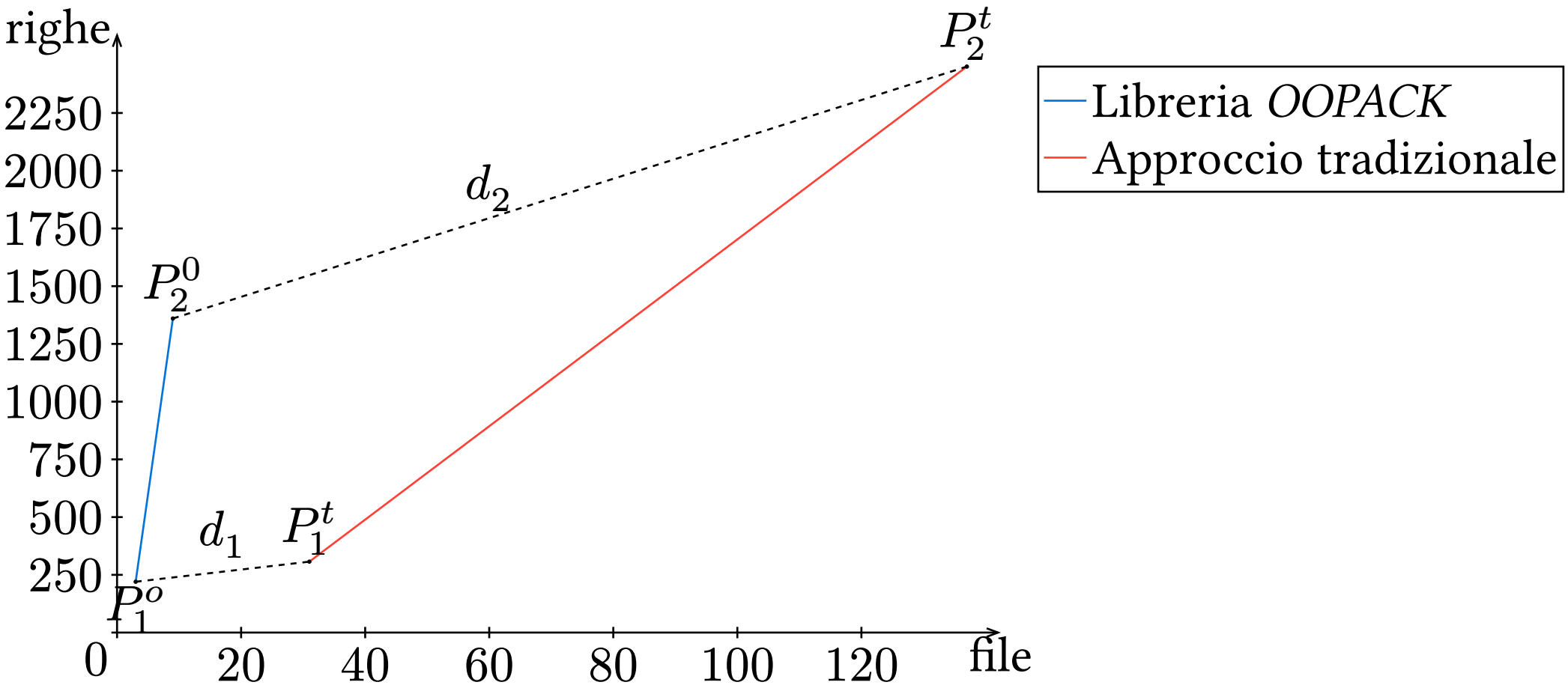


Metodi di utilità

Tramite il metodo `find()` è possibile ottenere riferimenti a file specifici da qualsiasi punto del progetto o di istanziarne uno nuovo qualora esso non esista.
Questo, oltre a rappresentare un esempio di design pattern *lazy loading*, permette di definire metodi di alto livello quali `addTranslation(key, value)` e `addSound()` per inserire coppie chiave valore nell risorse dedicate alla localizzazione e registrazione di suoni.
Sempre sfruttando questo metodo è possibile dichiarare le funzioni da eseguire ogni *game loop* o ad ogni ricarica del progetto.

Working Example

Risultati e metriche



Calcolando il rapporto tra il numero di file impiegati, risulta evidente il vantaggio dell'utilizzo della libreria. Per il *working example* P_1 si nota che ogni file sorgente genera circa $\frac{31}{3} = 10,3$ file di output. Eseguendo la medesima operazione per P_2, di dimensioni maggiori, si ottiene invece $\frac{137}{9} = 15,2$.

Da questi valori si può dedurre che maggiore è la portata del progetto, più elevata è la quantità di file gestita automaticamente per ogni singola unità di codice scritta dallo sviluppatore.

Interpretando la distanza tra il punto di partenza (sorgente) e quello di arrivo (output) come una stima del carico di lavoro automatizzato dalla

libreria, è evidente che automatizzare lo sviluppo sia vantaggioso per i progetti di scala maggiore.

Per il progetto minore P_1, la distanza è $d_1 = 91,4$. Per il progetto maggiore P_2, tale valore sale a $d_2 = 1098,5$.

Se si misura la densità di codice del singolo progetto p, come il rapporto tra le sue righe totali e file totali, si vedrà che $p(P_1) = 73,7$ e $p(P_2) = 151,1$.
Confrontando questi due valori si nota che a fronte di un raddoppio della densità nel progetto più grande ($p(P_2) \approx 2 \cdot p(P_1)$), il beneficio dell'automazione d cresce di un fattore 12 ($\frac{d_2}{d_1} \approx 12$).

Grazie per l'attenzione