# Object Oriented Pack Creation

## Introduzione

Per modificare il gioco Minecraft, è disponibile un linguaggio di programmazione (una *domain specific language*), chiamato *mcfunction*. In mcfunction si eseguono gruppi di comandi, uno dopo l'altro, contenuti in file detti funzioni. Un comando è l'unità fondamentale per modificare o aggiungere un qualche comportamento.
Questo linguaggio, assieme a file json, png e ogg fornisce a utenti, non necessariamente esperti di programmazione, un modo per modificare il gioco. In genere tutto ciò che è mcfunction modifica qualche comportamento, e tutto ciò che è json aggiunge qualche feature.

## Problemi

Quando si lavora su progetti di grandi dimensioni, ci si trova a dover gestire una grande quantità di file per un numero sproporzionatamente piccolo di feature. Dato che ogni funzione deve risiedere in un suo file `.mcfunction`, e non ci sono i classici *code blocks* di linguaggi come C o Java, bisogna creare una nuova funzione (ovvero un nuovo file) ogniqualvolta si vuole fare un `if` o `for`[1].
L'impossibilità di dichiarare più "oggetti" (es. funzioni, file json) in un unico file porta ad ambienti di lavoro difficili da navigare. Questo è esacerbato dalla già citata necessità di creare nuovi file ogni volta che si deve eseguire condizionalmente un gruppo di comandi.

Questa è la struttura (immutabile, imposta così dal compilatore) per implementare un oggetto (*item* del gioco) chiamato bar[2].

```
project root
├── datapack
│   └── data
│       └── foo
│           ├── loot_table
│           │   └── bar.json
│           ├── recipe
│           │   └── bar.json
│           └── function
│               └── bar
│                   ├── main.mcfunction
│                   ├── conditional.mcfunction
│                   └── other_conditional.mcfunction
└── resourcepack
    └── assets
        └── foo
            ├── lang
            │   └── en_us.json
            ├── models
            │   └── item
            │       └── bar.json
            └── textures
                └── item
                    └── bar.png
```

[1] mcfunction non dispone di un classico ciclo `for` ma può essere ricreato con la ricorsione.

[2] Tra le varie feature che si possono aggiungere, un item è la più semplice, e nonostante ciò richiede **almeno** 6 file per gestire pochi comportamenti personalizzati.

- `datapack`: contiene i file che modificano comportamenti (funzioni, ricette, bottino, obiettivi,...).
- `resourcepack`: contiene le risorse (suoni, font, modelli 3d, texture, traduzioni,...)
- `foo`: ogni progetto deve usare (almeno) un namespace[3] per distinguersi dalle feature degli altri ed evitare conflitti.
- `loot_table` e `recipe`: alcuni dei file json utilizzati per aggiungere feature. In questo caso `loot_table` contiene i dati che dovrà avere il mio oggetto (nome, rarità, danno) e `recipe` definisce gli ingredienti per crearlo.
- `lang/en_us`: traduzione in inglese del nome dell'oggetto
- `models/item` e `textures/item`: aspetto dell'oggetto
- `function`: comportamenti dell'oggetto (ad esempio cosa fa se clicco con il tasto destro quando lo impugno).

Come si può vedere bisogna gestire molti file (anche lontani tra loro) per definire per bene tutte le proprietà e comportamenti di una feature estremamente semplice.

Ogni progetto dispone di un *datapack*, che influenza il comportamento, e una *resourcepack*, che definisce le risorse utilizzate dal gioco. Questa è una struttura molto simile a un tipico progetto java con maven, dove il codice sorgente è separato dalle risorse. La cartella contenente *datapack* e *resourcepack*, rappresenta il *pack*, il cui nome è in genere quello del progetto[4].

Nella gerarchia dell'esempio, le cartelle corrispondenti a *datapack* e *resourcepack* sono situate in una cartella radice esterna a `.../.minecraft`. Per essere lette correttamente dal compilatore, queste devono per forza essere in specifiche sottocartelle di `.../.minecraft`. Dato questi path sono relativamente distanti, lo sviluppo *datapack* e *resourcepack* in contemporanea in un ambiente unificato è impossibile[5]. Per ovviare a questo problema si usano `junction` o `symbolic link` per creare alias delle cartelle sorgenti in una cartella unica. Questo consente anche di caricare progetti su GitHub in un unica repository contenente sia la parte di codice che quella delle risorse.

Un esempio di progetto completo può essere visto qua: https://github.com/asdru22/CognitionDev/

## La soluzione

Adottare un sistema basato su oggetti dove sono gia disposti metodi per creare questi file nei punti giusti e inserire i loro contenuti facilmente. Una possibile struttura potrebbe essere trattare tutti i file (mcfunction, json) come oggetti, dove il loro *filename* è generato automaticamente (ad esempio un uuid esadecimale casuale) quando si fa `build` del progetto, e la rappresentazione in forma di stringa di quel file corrisponde al path usato per invocarlo.

```
// java 21+
{
  void main(){
    Project myProject = new Project();

    Namespace myNamespace = new Namespace("foo");

    Datapack myDatapack = new Datapack();
    Resourcepack myResourcepack = new Resourcepack();
```

[3] Il namespace in genere coincide con il nome del progetto, o un suo acronimo.

[4] *pack* e progetto sono termini equivalenti e indicano la stessa cosa.

[5] I *datapack* vanno collocati in `.../.minecraft/saves/<world name>/datapacks/`, mentre le *resourcepack* in `.../.minecraft/resourcepacks/`

```
    Function fun2 = new Function("say I'm fun 2")

    Function fun1 = new Function(STR."""
        say hello world
        function \{fun2}
      """)

    myDatapack.add(fun1,fun2) // esempio grossolano, implementazione da
migliorare
    myResourcepack.add(...)

    myProject.setDatapack(myDatapack);
    myProject.setResourcepack(myResourcepack);

    myProject.setVersion("25w19a");
    myProject.setOutput("C:\Users\...\.minecraft");
    myProject.setWorld("Test World");

    myProject.build();

  }
}
```

Ad esempio `System.out.println(fun1)` restituirebbe `foo:c1a3e5d` e il file `c1a3e5d.mcfunction` conterrebbe il testo

```
say hello world
function foo:e069dc4 // rappresenta l'oggetto fun2
```

In questo modo l'utente non deve preoccuparsi di imparare il path della funzione o altri dati, e può semplicemente utilizzare la variabile corrispondente, il cui metodo `toString()`, chiamato quando si fa *build* del progetto, inserirà il path giusto nel file.

Oltre a permettere di scrivere le chiamate a funzioni/file json in questo modo, basare l'intero progetto su java permetterebbe anche di scrivere righe di codice ripetuto in maniera più veloce. Ad esempio se voglio eseguire un controllo di tutti gli slot dell'inventario di un giocatore dovrei scrivere

```
execute if items entity @s inventory.0 stone_sword run function foo:bar
execute if items entity @s inventory.1 stone_sword run function foo:bar
execute if items entity @s inventory.2 stone_sword run function foo:bar
...
execute if items entity @s inventory.35 stone_sword run function foo:bar
```

Questo può essere semplificato con

```
StringBuilder sb = new StringBuilder();
        for(int i = 0 ; i < 36 ; i++) {
            sb.append(String.format("execute if items entity @s inventory.%s
stone_sword run function foo:bar",i));
        }
new Function(sb.toString());
```

Dato che non esistono funzioni $\sin, \cos, \log$ etc... si usano delle lookup table che contengono dei valori in un range prefissato. In genere si creano con python e poi si fa copia-incolla in una funzione. Tuttavia con questo approccio si può semplificare con

```
StringBuilder sb = new StringBuilder();
    for(int i = 0; i<=360; i++){
        sb.append(STR."\{Math.sin(Math.toRadians(i))},");
    }
new Function(STR."data modify storage foo:storage root.sin set value [\{sb}]");
```

e poi prelevare dall'array l'elemento $i$ che corrisponde a $\sin(i)$[6].

La scrittura di costrutti `if` può essere semplificata con

```
Function f1 = new Function(STR."""
    execute if score foo.test matches 0 run say hi
    execute if score foo.test matches 1 run function \{new Function("""
      say command 1
      say command 2
      say command 3
      """)}
""");
```

che poi verrà compilato in

```
// foo:c1a3e5d.mcfunction
execute if score foo.test matches 0 run say hi
execute if score foo.test matches 1 run function foo:be5c05e

// foo:be5c05e.mcfunction
say command 1
say command 2
say command 3
```

Similmente per la ricorsione

```
//crea una linea di fuoco lunga 10 blocchi
Function raycast = new Function("""
  execute if score foo.test matches 0 run return fail
  scoreboard players remove foo.test 1
  particle fire
  execute positioned ^ ^ ^1 run function \{raycast}
""")

Function f1 = new Function(STR."""
    scoreboard players set foo.test 10
    function \{raycast}
""");
```

diventa

[6] Ad esempio `data get storage foo:storage root.sin[90]` restituirà 1

```
// foo:c1a3e5d.mcfunction
scoreboard players set foo.test 10
function foo:be5c05e

// foo:be5c05e.mcfunction
execute if score foo.test matches 0 run return fail
scoreboard players remove foo.test 1
particle fire
execute positioned ^ ^ ^1 run function foo:be5c05e
```

Dato che l'utente non interagirà mai con i file generati, non li interessa la struttura e quindi tutte le funzioni potrebbero potenzialmente essere nella stessa cartella radice[7]. Questo ha il leggero vantaggio di ridurre l'overhead per le chiamate di funzioni, il quale aumenta con il numero di sottocartelle.

## Plugin Gradle

Per rendere il progetto più interessante, si potrebbe creare un plugin Gradle che permette di

- esportare *datapack* e *resourcepack* in una cartella specifica su github tramite *actions* e facilitare la pubblicazione di versioni diverse del progetto.
- fare "refresh" delle versioni di Minecraft disponibili per il progetto, in modo da poter usare le nuove feature del gioco senza dover aggiornare manualmente il file contenente le `enum`.
- copiare i file relativi alle strutture (in formato `.nbt`) dal mondo in cui vengono costruite alla cartella risorse del progetto.

## Conclusione

L'attuale ambiente di sviluppo presenta molti problemi e limitazioni, sia a livello di struttura che a livello sintattico. Le limitazioni sintattiche o di feature realizzabili per molti sviluppatori rappresenta una sfida benvenuta, che fa parte del fascino di mcfunction. Per quanto riguarda la struttura, personalmente spesso mi ritrovo a non finire mai progetti perché ho raggiunto una soglia dove diventa troppo lungo e difficile navigare questi file. Mi piacerebbe dunque proporre questa soluzione al problema.

La struttura proposta, basata su oggetti, secondo me può non solo fornire un ambiente di lavoro più chiaro e facile da navigare, ma anche velocizzare la scrittura di codice, specialmente per quanto riguarda l'inserimento dinamico di testo in altri file. Ad esempio si potrebbe implementare un metodo `addTranslation(String key, String value)` che consente di inserire un'entry al file json delle traduzioni in inglese[8] da qualsiasi punto del progetto. Oppure usare un'annotazione per definire quali funzioni sono eseguite costantemente[9] (`@Tick`). Un altro vantaggio notevole di basare questi progetti su un linguaggio di programmazione è la possibilità di integrare feature di java ovunque un utente lo ritenga utile per rendere mcfunction un linguaggio più di alto livello (possibilità di ripetere facilmente codice, generare template per creare oggetti facilmente sfruttando l'ereditarietà, creare *lookup table* in maniera più coincisa…).

Secondo me una volta finito questo progetto può essere condiviso ed essere utilizzato come libreria da altri utenti che hanno riscontrato gli stessi problemi con mcfunction. Per questo un possibile lavoro da fare può anche essere creare una documentazione con javadoc, progettare le classi

[7] Ad esempio tutti i *datapack* saranno nella cartella `foo/function/`

[8] `en_us.json` è la traduzione di default e fallback per tutte le traduzioni assenti in altre lingue.

[9] Il ciclo di gioco di Minecraft normalmente scorre a una velocità fissa di 20 tick al secondo, cioè c'è un tick ogni 0,05 secondi. Quindi una funzione in "loop" viene eseguita 20 volte al secondo.

ponendo particolare attenzione alla visibilità di metodi e campi e quali classi possano essere a disposizione degli utenti, e fornire messaggi di errore (ad esempio se si cerca di chiamare `build` su un oggetto `Project` dove non è stato dichiarato un *datapack*).[10] Inoltre, conterei di applicare design pattern per ridurre le ripetizioni nel codice (specialmente considerando le similitudini e differenze tre *datapack* e *resourcepack*), rendere il codice facilmente modificabile in futuro e leggibile.

## Conoscenze acquisibili e applicabili durante lo sviluppo

- Utilizzo dei già citati design pattern (visti ad ingegneria del software):
- Utilizzo API di java moderni per operare su collezioni di oggetti come `Collections` e `Stream`;
- Sviluppo e pubblicazione una libreria Gradle (kotlin);
- Implementazione di strutture dati complesse

[10] Per esempio se si chiama `myProject.build()` senza aver prima chiamato `myProject.setDatapack(myDatapack)` il programma dovrebbe lanciare un'eccezione.