Object Oriented Pack Creation

Introduzione

Per modificare il gioco Minecraft, è disponibile un linguaggio di programmazione (una *domain specific language*), chiamato *mcfunction*[1]. 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

Il problema emerge quando si lavora su progetti di grandi dimensioni, dove 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`[2]. L'impossibilità di dichiarare più "oggetti" (sia funzioni, sia 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[3].

```
project
├── 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] Non sto facendo riferimento alle cosiddette mod, che sono scritte in Java e modificano il codice sorgente.

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

[3] 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[4] 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 del mio oggetto
- `function` : comportamenti del mio 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[5].

La struttura mostrata in esempio pone *datapack* e *resourcepack* in una cartella esterna a quella di `.minecraft` . Normalmente queste due cartelle devono per forza essere in specifici path locali per essere letti correttamente dal compilatore. Tuttavia, questi path sono relativamente distanti e renderebbe sviluppare *datapack* e *resourcepack* in contemporanea molto complicato[6]. Per ovviare a questo problema si usano `junction` o `symbolic link` per creare alias delle cartelle sorgenti nei path corretti. 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 inserire questi file e generare i progetti. Una possibile struttura potrebbe essere trattare tutti i file come oggetti, dove il loro nome è generato automaticamente (ad esempio un uuid esadecimale casuale), 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();


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

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

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

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

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

    myDatapack.add(...)
    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(fun2)` restituirebbe `foo:bcfa538a-72d4-4996-8639-527e26abbcb1` e il file `fun1.mcfunction` conterrebbe il testo

```
say hello world
function foo:bcfa538a-72d4-4996-8639-527e26abbcb1
```

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 copia 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 $\sin(i)$[7].

Alternativa

Un metodo alternativo può consistere nel sfruttare le annotazioni, e poi tramite reflection ottenere il loro nome e il contenuto del campo associato.

```
@Folder(name="foo")
public class Test {

    @Function(name = "myfunction")
    String f1 = STR."""
            say hi
            function \{Global.getNamespace()}:foo/another_function
            """;

    @Function(name = "another_function")
    String f2 = "say im a function";

    String f3 = "not a function";
}
```

Questo creerà per esempio `namespace:foo/myfunction.mcfunction`, e il testo al suo interno sarà

```
say hi
function namespace:foo/another_function
```

Conclusione

L'attuale ambiente di sviluppo presenta molti problemi e limitazioni, sia a livello di struttura che a livello sintattico. Le limitazioni per molti sviluppatori rappresentano una sfida e ormai sono in grado di sviluppare feature aggirandole strategicamente. 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 aggiunge un entry al file json della lingua inglese[8]. 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

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

[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.

rendere mcfunction un linguaggio più di alto livello (possibilità di ripetere facilmente codice, generare template per creare oggetti facilmente, creare *lookup table* velocemente…).

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 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*).

Ci tengo a precisare che per me l'aspetto prioritario della proposta resta la programmazione ad oggetti, utilizzare feature più avanzate di java e design pattern di ingegneria software. Quindi se lei preferisce un altro scenario (non lo sviluppo di pack per Minecraft) in cui è possibile fare un lavoro simile per me non c'è problema.

Invece se le piace il progetto presentato, preciso che gli esempi forniti sono solo la punta del iceberg di una serie di feature che si potrebbero implementare con un sistema basato interamente su oggetti per velocizzare lo sviluppo, piuttosto che utilizzare il classico sistema basato sulla creazione manuale di file. Dato che ho usato questi pack per più di 7 anni, so bene quali sono i suoi punti deboli, e feature che la comunità vorrebbe.

Secondo me è richiesto un certo impegno per sviluppare questo sistema basato su oggetti in maniera completa, funzionale, ben documentata e con gestione degli errori. Auspico quindi che la presente proposta di tesi possa proseguire anche attraverso un tirocinio interno, così da poterne sviluppare in modo più completo sia l'implementazione tecnica sia gli aspetti pratici.