ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI SCIENZA E INGEGNERIA

Corso di Laurea in Informatica per il Management

Un Framework per la Meta-programmazione di Minecraft

Relatore:
Prof. Luca Padovani

Presentata da:
Alessandro Nanni

Sessione di Dicembre
Anno accademico 2024/2025

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI SCIENZA E INGEGNERIA

Corso di Laurea in Informatica per il Management

Un Framework per la Meta-programmazione di Minecraft

Relatore:
Prof. Luca Padovani

Presentata da:
Alessandro Nanni

Sessione di Dicembre
Anno accademico 2024/2025

Sommario

La *domain specific language* (DSL) di Minecraft, denominata *mcfunction*, consente la creazione di pacchetti di contenuti modulari, denominati «*pack*», in grado di modificare o aggiungere meccaniche di gioco. Nonostante il suo ampio utilizzo, questo linguaggio presenta notevoli limitazioni strutturali e sintattiche: ogni funzione deve essere definita in un file separato e mancano costrutti di programmazione come variabili, istruzioni condizionali e meccanismi di iterazione. Questi vincoli producono codice prolisso e ripetitivo, compromettendo la leggibilità e la manutenibilità nei progetti di ampia scala.

Per superare tali problemi, questa tesi propone una libreria Java sviluppata durante il tirocinio accademico, a partire da un'analisi approfondita delle carenze e difetti di *mcfunction* e giungendo alla formulazione di un'astrazione che rappresenta la struttura di un *pack* come un albero di oggetti tipizzati. Sfruttando la sintassi standard di Java e *factory methods*, la libreria consente la generazione programmatica dei *pack*, offrendo zucchero sintattico e utilità che semplificano l'accesso ai file di risorse principali. L'approccio proposto fornisce validazione in fase di compilazione, supporta la definizione di più risorse all'interno di un singolo file sorgente e automatizza la generazione di *boilerplate*, eliminando così la necessità di preprocessori esterni o di sintassi ibride adottate da soluzioni alternative.

Un *working example* valida l'approccio: il *pack* di esempio richiede il 40% di codice in meno, consolidando 31 file in 3 file sorgenti, e dimostra miglioramenti significativi in termini di densità del codice e manutenibilità del progetto.

Desidero ringraziare il professor Padovani per la disponibilità e il prezioso supporto a me offerto durante questo percorso. Lo ringrazio anche per avermi dato l'opportunità di approfondire e lavorare con tecnologie a me particolarmente care.

Indice dei contenuti

Sommario **1**

Indice dei contenuti **3**

1. Introduzione **4**

2. Struttura e Funzionalità di un Pack **7**

2.1. Cos'è un Pack 7
2.2. Struttura e Componenti di Datapack e Resourcepack 8
2.3. Comandi 10
2.4. Funzioni 13

3. Problemi Pratici e Limiti Tecnici **15**

3.1. Limitazioni di Scoreboard 16
3.2. Assenza di Funzioni Matematiche 17
3.3. Alto Rischio di Conflitti 19
3.4. Assenza di Code Blocks 20
3.5. Organizzazione e Complessità della Struttura dei File 23
3.6. Stato dell'Arte delle Ottimizzazioni del Sistema 25

4. Descrizione della Libreria **27**

4.1. Approccio al Problema 27
4.2. Classi Astratte e Interfacce 30
4.3. Classi Concrete 38
4.4. Utilità 41
4.5. Uso working example 45

5. Conclusione **51**

Bibliografia **54**

Introduzione

Se non fosse per il videogioco *Minecraft* [1], non sarei qui ora. Quello che per me nel 2014 era un modo di esprimere la mia creatività costruendo con cubi in un mondo tridimensionale, si è rivelato presto essere il luogo dove per anni ho scritto ed eseguito i miei primi frammenti di codice tramite il suo sistema di comandi.
Motivato dalla mia abilità nel saper programmare con questo linguaggio di scripting non convenzionale, ho perseguito una carriera di studio in informatica.

Pubblicato nel 2012 dall'azienda svedese Mojang [2], *Minecraft* è un videogioco appartenente al genere *sandbox* [3], famoso per l'assenza di una trama predefinita, in cui è il giocatore stesso a costruire liberamente la propria esperienza e gli obiettivi da perseguire.
Come suggerisce il nome, le attività principali consistono nello scavare per ottenere risorse e utilizzarle per creare nuovi oggetti o strutture. Il tutto avviene all'interno di un ambiente tridimensionale virtualmente infinito.

Proprio a causa dell'assenza di regole predefinite, fin dal suo rilascio *Minecraft* era dotato di un insieme rudimentale di comandi [4] che consentiva ai giocatori di aggirare le normali meccaniche di gioco, ad esempio ottenendo risorse istantaneamente o spostandosi liberamente nel mondo.
Con il tempo, tale meccanismo è diventato un articolato linguaggio di configurazione e scripting, basato su file testuali, che costituisce una *Domain Specific Language* [5] (DSL) attraverso la quale sviluppatori di terze parti possono modificare numerosi aspetti e comportamenti dell'ambiente di gioco.

Con *Domain Specific Language* si intende un linguaggio di programmazione progettato per un ambito applicativo specifico, caratterizzato da un livello di astrazione più elevato e una sintassi semplificata rispetto ai linguaggi *general purpose*[1]. Le DSL sono sviluppate in coordinazione con esperti del campo nel quale verrà utilizzato il linguaggio.

> In many cases, DSLs are intended to be used not by software people, but instead by non-programmers who are fluent in the domain the DSL addresses.
>
> — JetBrains

Questa definizione fornita dagli sviluppatori di JetBrains, azienda specializzata nello sviluppo di ambienti di sviluppo integrati (IDE), descrive perfettamente chi sono gli utilizzatori della *domain specific language* di *Minecraft*.

Minecraft è sviluppato in Java [6], ma questa DSL, chiamata *mcfunction* [7], adotta un paradigma completamente diverso. Essa non consente di introdurre nuovi comportamenti intervenendo direttamente sul codice sorgente del gioco. Le funzionalità aggiuntive vengono invece definite attraverso gruppi di comandi testuali, interpretati dal motore interno di *Minecraft* (e non dal compilatore Java) ed eseguiti solo al verificarsi di determinate condizioni. In questo modo l'utente percepisce tali funzionalità come parte integrante dei contenuti originali del gioco. Negli ultimi anni, grazie all'introduzione e all'evoluzione di file in formato JSON [8] in grado di modificare componenti precedentemente inaccessibili, è progressivamente diventato possibile creare esperienze di gioco quasi completamente nuove. Tuttavia, il sistema presenta ancora diverse limitazioni, poiché una parte sostanziale della logica continua a essere implementata attraverso i file *mcfunction*.

Il tirocinio ha avuto come obiettivo la progettazione e realizzazione di un framework che semplifica lo sviluppo e la distribuzione di questi file tramite un ambiente di sviluppo unificato. Esso consiste in una libreria Java che permette di definire la gerarchia dei file in un sistema ad albero tramite oggetti. Una volta definite tutte le *feature*, viene eseguito il programma per ottenere un progetto pronto per essere utilizzato.

In questo modo lo sviluppo risulta più coerente e accessibile, permettendo di integrare *feature* di Java in questa DSL, per facilitare la scrittura e gestione dei file.

Nel capitolo successivo viene presentata la struttura generale del sistema di *pack*, descrivendone gli elementi che lo costituiscono e come essi funzionano. Segue un'analisi sistematica delle principali problematiche e limitazioni tecniche dell'infrastruttura, corredata da una rassegna critica delle soluzioni proposte nello stato dell'arte. Viene quindi illustrata la progettazione e l'implementazione della libreria sviluppata, accompagnata da un caso d'uso concreto (*working example*) che ne dimostra l'applicazione pratica. Il lavoro si conclude con un'analisi

[1] Un linguaggio *general purpose* (o «a scopo generale») è progettato per risolvere un'ampia varietà di problemi in diversi domini applicativi.

quantitativa e qualitativa dei risultati ottenuti, evidenziando i benefici dell'approccio proposto in termini di riduzione della complessità e miglioramento della manutenibilità del codice.

Struttura e Funzionalità di un Pack

2.1. Cos'è un Pack

Affinché i file JSON e *mcfunction* vengano riconosciuti dal compilatore di *Minecraft* e integrati nel videogioco, è necessario che siano collocati in specifiche *directory* predefinite.
Un *datapack* può essere paragonato alla cartella `java` di un progetto Java. Esso contiene la parte che detta la logica dell'applicazione.

I progetti Java sono dotati di una cartella `resources` [9]. Similmente, *Minecraft* dispone di una cartella in cui dichiarare le risorse. Questa si chiama *resourcepack* [10], e contiene principalmente font, modelli 3D, *texture* [11], traduzioni e suoni.
Con l'eccezione di *texture* e suoni, i quali richiedono l'estensione `png` [12] e `ogg` [13] rispettivamente, tutti gli altri file sono in formato JSON.
Le *resourcepack* sono state concepite e rilasciate prima dei *datapack*, con lo scopo di dare ai giocatori un modo di sovrascrivere le *texture* e altri *asset* [14] del videogioco per renderle più affine ai propri gusti. Gli sviluppatori di *datapack* hanno poi iniziato ad utilizzare *resourcepack* per definire le risorse che loro il progetto avrebbe richiesto. I *resourcepack* hanno portata

globale e vengono applicati a tutti i *save file*, ovvero su ogni mondo attualmente in uso. Le cartelle *datapack*, invece, devono essere collocate nella directory `datapack` dei singoli mondi nei quali si desidera utilizzarle.
Pertanto, partendo dalla cartella radice di *Minecraft* (`.minecraft/`), i *resourcepack* si trovano nella directory `.minecraft/resourcepacks`, mentre i *datapack* sono posizionati in `.minecraft/saves/<world name>/datapacks`.
L'insieme di *datapack* e *resourcepack* è chiamato *pack*. Questo, riprendendo il parallelismo precedente, corrisponde all'intero progetto Java, e sarà poi la cartella che lo sviluppatore pubblicherà.

2.2. Struttura e Componenti di Datapack e Resourcepack

All'interno di un *pack*, *datapack* e *resourcepack* hanno una struttura molto simile.

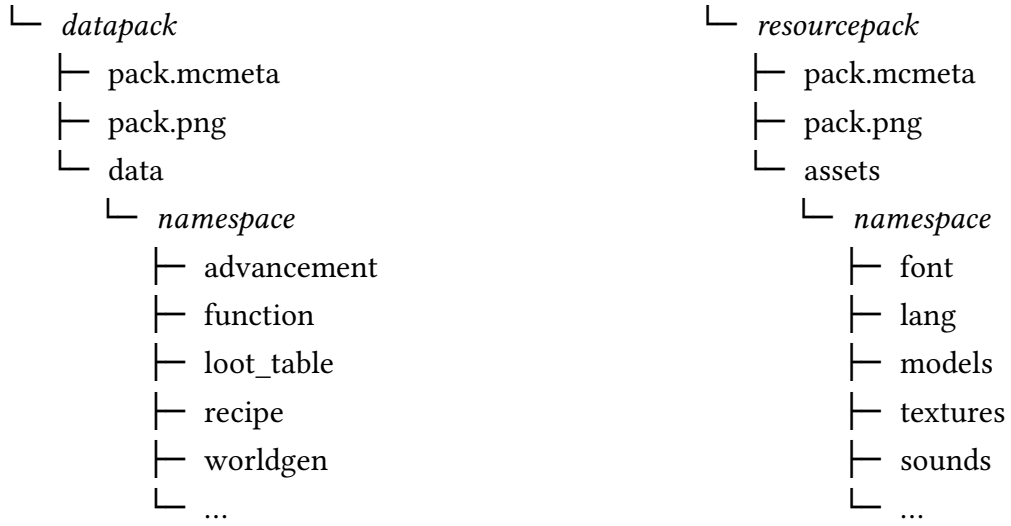

Figura 1: *datapack* e *resourcepack* a confronto.

Nonostante l'estensione non lo indichi, il file `pack.mcmeta` è scritto in formato JSON e definisce l'intervallo delle versioni (denominate *format*) supportate dalla cartella. Tali versioni variano ad ogni aggiornamento di *Minecraft* e non corrispondono alla *game version* effettiva. Ad esempio, per la versione 1.21.10 del gioco, il `pack_format` dei *datapack* è 88, mentre quello delle *resourcepack* è 69. Questi valori possono variare anche settimanalmente durante il rilascio degli *snapshot* [15], ovvero versioni preliminari di sviluppo che introducono nuove funzionalità e modifiche prima del rilascio ufficiale di un aggiornamento.

Ancora più rilevanti sono le cartelle contenute in `data` e `assets`, chiamate *namespace* [16]. Se i progetti Java seguono la struttura `com.package.author`, allora i *namespace* possono essere visti come la sezione `package`.

> This isn't a new concept, but I thought I should reiterate what a «namespace» is. Most things in the game has a namespace, so that if we add `something` and a mod (or map, or whatever) adds `something`, they're both different `something`s. Whenever you're asked to name something, for example a loot table, you're expected to also provide what namespace that thing comes from. If you don't specify the namespace, we default to `minecraft`. This means that `something` and `minecraft:something` are the same thing.
>
> — Nathan Adams[2]

I *namespace* sono fondamentali per evitare che i file omonimi di un *pack* sovrascrivano quelli di un altro. Per questa ragione in genere i *namespace* o sono abbreviazioni o coincidono con il nome stesso del progetto che si sta sviluppando, e si usa lo stesso tra *datapack* e *resourcepack*. Tuttavia, si vedrà come operare in *namespace* distinti non sia sufficiente a garantire l'assenza di conflitti tra *pack* installati contemporaneamente.

Il namespace `minecraft` è riservato alle risorse native del gioco: sovrascriverle comporta il rischio di rimuovere funzionalità originali o di alterare il comportamento previsto del gioco. È interessante notare come anche gli sviluppatori di *Minecraft* stessi facciano uso dei *datapack* per definire e organizzare molti comportamenti del gioco, come la dichiarazione delle risorse ottenibili da un baule, o gli ingredienti necessari per creare un certo oggetto. In altre parole, i *datapack* non sono solo uno strumento a disposizione dei giocatori per personalizzare l'esperienza, ma costituiscono anche il meccanismo interno attraverso cui il gioco stesso struttura e gestisce alcune delle sue funzionalità principali.
Bisogna specificare che i comandi e file `.mcfunction` non sono utilizzati in alcun modo dagli sviluppatori di *Minecraft* per implementare funzionalità del videogioco. Come precedentemente citato, tutta la logica è dettata da codice Java.

All'interno dei *namespace* si trovano directory i cui nomi identificano in maniera univoca la natura e la funzione dei contenuti al loro interno. Se è presente un file JSON nella cartella `recipe`, che non possiede una struttura comune a tutte le ricette, il compilatore solleverà un errore e il file non sarà disponibile nella sessione di gioco.

In `function` si trovano file e sottodirectory contenenti file di testo in formato *mcfunction*. Questi si occupano di far comunicare le parti di un *pack* tra loro tramite funzioni contenenti comandi.

[2] Sviluppatore di *Minecraft* parte del team che implementa *feature* inerenti a *datapack*.

2.3. Comandi

Prima di spiegare cosa fanno i comandi, è necessario definire gli elementi basilari su cui essi agiscono.
Minecraft permette di creare ed esplorare mondi generati a partire da un *seed* [17] casuale. Ogni mondo è composto da *chunk* [18], sezioni colonnari aventi base di 16×16 unità e altezza di 320 unità.
L'unità più piccola all'interno di questa griglia è il blocco, la cui forma corrisponde a quella di un cubo di lato unitario. Ogni blocco è dotato di collisione, ed individuabile in un mondo tramite coordinate dello spazio tridimensionale. Si definiscono entità invece tutti gli oggetti dinamici che si spostano in un mondo: sono dotate di una posizione, rotazione e velocità.

I dati persistenti di blocchi ed entità sono compressi e memorizzati in una struttura dati ad albero chiamata *Named Binary Tags* [19] (NBT). Il formato «stringificato», `SNBT` è accessibile agli utenti e si presenta come una struttura molto simile a JSON, formata da coppie di chiave e valori.

```snbt
{
  name1: 123,
  name2: "foo",
  name3: {
    subname1: 456,
    subname2: "bar"
    },
  name4: [
    "baz",
    456,
    {
      subname3: "bal"
    }
  ]
}
```

Codice 1: Esempio di `SNBT`.

Un comando è un'istruzione testuale che Minecraft interpreta per eseguire una specifica azione, come assegnare oggetti al giocatore, modificare l'ora del giorno o creare entità. Molti comandi usano selettori per individuare l'entità su cui essere applicati o eseguiti.

```mcfunction
say @e[
  type = player
]
```

Codice 2: Esempio di comando che tra tutte le entità (`@e`), stampa quelle di tipo giocatore.

Sebbene il sistema dei comandi sia privo delle funzionalità tipiche dei linguaggi di programmazione di alto livello, quali cicli `for` e `while`, strutture dati complesse o variabili generiche, esso fornisce comunque strumenti che consentono di emulare alcuni di questi comportamenti in forma limitata. Di seguito verranno illustrati i comandi che più si avvicinano a concetti tipici di programmazione.

2.3.1. Scoreboard

Il comando `scoreboard` permette di creare dizionari di tipo `<Entità, Objective>`. Un `objective` rappresenta un valore intero a cui è associata una condizione (*criteria*) che ne determina la variazione. Il *criteria* `dummy` corrisponde ad una condizione vuota, irrealizzabile. Su questi valori è possibile eseguire operazioni aritmetiche semplici, quali la somma o la sottrazione di un valore prefissato, oppure le quattro operazioni aritmetiche fondamentali[3] con altri `objective`. Dunque una *scoreboard* può essere meglio vista come un dizionario `<Entità,<Intero, Condizione>>`.
Prima di poter eseguire qualsiasi operazione su di essa, una *scoreboard* deve essere creata tramite il comando
`scoreboard objectives add <objective> <criteria>`.
Per eseguire operazioni che non dipendono da alcuna entità si usano i cosiddetti *fakeplayer*. Al posto di usare nomi di giocatori o selettori, si prefiggono i nomi con caratteri illegali, quali `$` e `#`. In questo modo ci si assicura che un valore non sia associato ad un vero utente, e quindi sia sempre disponibile.

```mcfunction
scoreboard objectives add my_scoreboard dummy
scoreboard players set #20 my_scoreboard 20
scoreboard players set #val my_scoreboard 100
scoreboard players operation #val my_scoreboard /= #20 my_scoreboard
```

Codice 3: Esempio di operazioni su una *scoreboard*, equivalente a `int val = 100; val /= 20;`

Dunque, il sistema delle *scoreboard* permette di creare ed eseguire operazioni semplici esclusivamente su interi, con *scope* globale, se e solo se fanno parte di una *scoreboard* dichiarata.

2.3.2. Data

Per ottenere, modificare e combinare i dati NBT associati a entità, blocchi e *storage* si usa il comando `data`. Come precedentemente citato, il formato NBT, una volta compresso, viene utilizzato per la persistenza dei dati di gioco. Oltre alle informazioni relative a entità e blocchi, in questo formato vengono salvati anche gli *storage*. Essi sono un modo efficiente di immagazzinare dati arbitrari senza dover dipendere dall'esistenza di un certo blocco o entità. Per prevenire i conflitti, ogni *storage* dispone di una *resource location*, che convenzionalmente coincide con il *namespace*. Vengono dunque salvati nel file `command_storage_<namespace>.dat` come dizionario NBT.

[3] Le operazioni aritmetiche fondamentali sono somma, sottrazione, moltiplicazione e divisione.

```mcfunction
data modify storage my_namespace:storage name set value "My Cat"
data merge entity @n[type=cat] CustomName from storage my_namespace:storage name
data remove storage my_namespace:storage name
```

Codice 4: Esempio di operazioni su dati NBT

Questi comandi definiscono la stringa `My Cat` nello *storage*, successivamente impostano il valore dallo *storage* al campo nome dell'entità gatto più vicina, e infine eliminano i dati dallo *storage*.

2.3.3. Execute

Il comando `execute` consente di eseguire un altro comando cambiando valori quali l'entità esecutrice e la posizione. Questi elementi definiscono il contesto di esecuzione: l'insieme dei parametri che determinano le modalità con cui il comando viene eseguito. Si usa il selettore `@s` per fare riferimento all'entità del contesto di esecuzione corrente.
Tramite `execute` è possibile specificare condizioni preliminari e salvare il risultato dell'esecuzione. Dispone di 14 sottocomandi, raggruppati in 4 categorie:

- modificatori: cambiano il contesto di esecuzione;
- condizionali: controllano se certe condizioni sono rispettate;
- contenitori: salvano i valori di output di un comando in una *scoreboard*, o in un contenitore di NBT;
- `run`: esegue un altro comando.

Tutti questi sottocomandi possono essere concatenati e usati più volte all'interno di uno stesso comando `execute`.

```mcfunction
execute as @e
  at @s
  store result score @s on_stone
  if block ~ ~-1 ~ stone
```

Codice 5: Esempio di comando `execute`.

Questo comando sta definendo quattro istruzioni da svolgere:

1. per ogni entità (`execute as @e`);
2. sposta l'esecuzione alla loro posizione attuale (`at @s`);
3. salva l'esito della prossima istruzione nello *score* `on_stone` di quell'entità;
4. controlla se nella posizione del contesto di esecuzione corrente, il blocco sottostante sia di tipo `stone`.

Al termine dell'esecuzione, lo *score* `on_stone` di ogni entità sarà 1 se si trovava su un blocco di pietra, 0 altrimenti.

2.4. Funzioni

Le funzioni sono insiemi di comandi raggruppati all'interno di un file *mcfunction*. Una funzione non può esistere se non in un file con estensione `.mcfunction`. A differenza di quanto il nome possa suggerire, non prevedono valori di input o di output, ma contengono uno o più comandi che vengono eseguiti in ordine.

In base alla complessità del branching e alle operazioni eseguite dalle funzioni, il compilatore (o più precisamente, il motore di esecuzione dei comandi) deve allocare una certa quantità di risorse per svolgere tutte le istruzioni durante un singolo *tick*. Il tempo di elaborazione aggiuntivo richiesto per l'esecuzione di un comando o di una funzione è definito *overhead*.

Ci sono più modi in cui le funzioni possono essere invocate da altri file di un datapack:

- tramite comandi: `function namespace:function_name` esegue la funzione immediatamente, mentre `schedule namespace:function_name <delay>` la esegue dopo un intervallo di tempo specificato;
- da *function tag*: una *function tag* è una lista in formato JSON contenente riferimenti a funzioni. *Minecraft* ne fornisce due nelle quali inserire le funzioni da eseguire rispettivamente ogni *game loop* [20](`tick.json`)[4], e ogni volta che si ricarica da disco il datapack (`load.json`). Queste due *function tag* sono riconosciute dal compilatore di *Minecraft* solo se nel namespace `minecraft` ;
- altre risorse di un *datapack* quali ricompense di `Advancement` (obiettivi) e effetti di `Enchantment` (incantesimi).

Le funzioni vengono eseguite durante un *game loop*, completando tutti i comandi che contengono, inclusi quelli invocati altre funzioni. Quando un comando `execute` altera il contesto di esecuzione, la modifica non influenza i comandi successivi, ma viene propagata alle funzioni chiamate a partire da quel punto.

Le funzioni possono includere linee *macro*: comandi che, preceduti dal carattere `$`, dispongono di una o più sezioni delimitate da `$(...)`, le quali vengono sostituite al momento dell'invocazione con oggetti NBT specificati nel comando invocante.

main.mcfunction `mcfunction`

```
function foo:macro_test {value:"bar"}
function foo:macro_test {value:"123"}
```

macro_test.mcfunction `mcfunction`

```
$say my value is $(value)
```

Codice 6: Esempio di chiamata di funzione con *macro*.

[4] Il *game loop* di *Minecraft* viene eseguito 20 volte al secondo; di conseguenza, anche le funzioni incluse nel tag `tick.json` vengono eseguite con la stessa frequenza.

Il primo comando di `main.mcfunction` stamperà `my value is bar`, il secondo `my value is 123`.

L'esecuzione dei comandi di una funzione può essere interrotta dal comando `return`. Funzioni che non contengono questo comando possono essere considerate di tipo `void`. Tuttavia il comando return può solamente restituire la parola chiave `fail` o un valore intero fisso.

Una funzione può essere richiamata ricorsivamente, anche modificando il contesto in cui viene eseguita. Questo comporta il rischio di creare chiamate senza fine, qualora la funzione sia invocata senza alcuna condizione di arresto. È quindi responsabilità del programmatore definire i vincoli alla chiamata ricorsiva.

iterate.mcfunction `mcfunction`

```
particle flame ~ ~ ~
execute if entity @p[distance=..10] positioned ^ ^ ^0.1 run function
foo:iterate
```

Codice 7: Esempio di funzione ricorsiva che crea una scia lunga 10 blocchi nella direzione dove il giocatore sta guardando.

Ogni volta che viene chiamata, questa funzione istanzia una piccola *texture* intangibile e temporanea(*particle* [21]) alla posizione associata al contesto di esecuzione. Successivamente controlla se è presente un giocatore nel raggio di 10 blocchi. In caso positivo sposta il contesto di esecuzione avanti di $\frac{1}{10}$ di blocco e si chiama nuovamente la funzione. Quando il sotto-comando `if` fallisce, ovvero non c'è nessun giocatore nel raggio di 10 blocchi, la funzione non sarà più eseguita.

Un linguaggio di programmazione si definisce Turing completo [22] se soddisfa tre condizioni fondamentali:

1. Presenta rami condizionali: deve poter eseguire istruzioni diverse in base a una condizione logica. Nel caso di *mcfunction*, ciò è realizzabile tramite il sotto-comando `if`.
2. È dotato di iterazione o ricorsione: deve consentire la ripetizione di operazioni. In questo linguaggio, tale comportamento è ottenuto attraverso l'utilizzo di funzioni ricorsive.
3. Permette la memorizzazione di dati: deve poter gestire una quantità arbitraria di informazioni. In *mcfunction*, ciò avviene tramite la manipolazione dei dati all'interno dei *storage*.

Pertanto, *mcfunction* può essere considerato a tutti gli effetti un linguaggio Turing completo. Tuttavia, come verrà illustrato nella sezione successiva, sia il linguaggio stesso sia il sistema di file su cui si basa presentano diverse limitazioni e inefficienze.

In particolare, l'implementazione di funzionalità relativamente semplici richiede un numero considerevole di righe di codice e di file, che in un linguaggio di più alto livello potrebbero essere realizzate in maniera molto più coincisa.

Problemi Pratici e Limiti Tecnici

Il linguaggio *mcfunction* non è stato originariamente concepito come un linguaggio di programmazione Turing completo. Ad esempio, nel 2012, prima dell'introduzione dei *datapack*, il comando `scoreboard` veniva utilizzato unicamente per monitorare statistiche dei giocatori, come il tempo di gioco o il numero di blocchi scavati. Gli sviluppatori di *Minecraft* osservarono come questo e altri comandi venivano impiegati dalla comunità per creare nuove meccaniche e giochi rudimentali, e hanno dunque aggiornato progressivamente il sistema, fino ad arrivare, nel 2017 alla nascita dei *datapack*.

Ancora oggi l'ecosistema dei *datapack* è in costante evoluzione, con *snapshot* che introducono nuove funzionalità o ne modificano di già esistenti. Tuttavia, il sistema presenta ancora diverse limitazioni di natura tecnica, riconducibili al fatto che non era stato originariamente progettato per supportare logiche di programmazione complesse o essere utilizzato in progetti di grandi dimensioni.

3.1. Limitazioni di Scoreboard

Come è stato precedentemente citato, `scoreboard` è usato per eseguire operazioni su interi. Tuttavia, l'utilizzo di questo comando presenta numerosi problemi.

Dopo che un *objective* è stato creato, è necessario impostare le costanti che si utilizzeranno, qualora si volessero eseguire operazioni di moltiplicazione e divisione. Inoltre, un singolo comando `scoreboard` prevede una sola operazione.

Di seguito viene mostrato come l'espressione `int x = (y*2)/4-2` si calcola in *mcfunction*. Le variabili saranno prefissate da `$`, e le costanti da `#`.

```mcfunction
scoreboard objectives add math dummy
scoreboard players set $y math 10
scoreboard players set #2 math 2
scoreboard players set #4 math 4
scoreboard players operation $y math *= #2 math
scoreboard players operation $y math /= #4 math
scoreboard players remove $y math 2
scoreboard players operation $x math = $y math
```

Operazioni su `$y`

Codice 8: Esempio con $y = 10$

Qualora non fossero stati impostati i valori di `#2` e `#4`, il compilatore li avrebbe valutati con valore 0 e l'espressione non sarebbe stata corretta.

Si noti come, nell'esempio precedente, le operazioni vengano eseguite sulla variabile y, il cui valore viene poi assegnato a x. Di conseguenza, sia `$x` che `$y` conterranno il risultato finale pari a 3. Questo implica che il valore di y viene modificato, a differenza dell'espressione a cui l'esempio si ispira, dove y dovrebbe rimanere invariato. Per evitare questo effetto collaterale, è necessario eseguire l'assegnazione $x = y$ prima delle altre operazioni aritmetiche.

```mcfunction
scoreboard objectives add math dummy
scoreboard players set $y math <some value>
scoreboard players set #2 math 2
scoreboard players set #4 math 4
scoreboard players operation $x math = $y math
scoreboard players operation $x math *= #2 math
scoreboard players operation $x math /= #4 math
scoreboard players remove $x math 2
```

Operazioni su `$x`

Codice 9: Esempio di espressione con `scoreboard`

La soluzione è quindi semplice, ma mette in evidenza come in questo contesto non sia possibile scrivere le istruzioni nello stesso ordine in cui verrebbero elaborate da un compilatore tradizionale.

Un ulteriore caso in cui l'ordine di esecuzione delle operazioni e il dominio ristretto agli interi assumono particolare rilevanza riguarda il rischio di errori di arrotondamento nelle operazioni che coinvolgono valori prossimi allo zero.

Si supponga si voglia calcolare il 5% di 40. Con un linguaggio di programmazione di alto livello si ottiene 2 calcolando `40/100*5` e `40*5/100`. Scomponendo queste operazioni in comandi `scoreboard` si ottiene rispettivamente:

```mcfunction
scoreboard players operation set $val math 40
scoreboard players operation $val math /= #100 math
scoreboard players operation $val math *= #5 math
```

```mcfunction
scoreboard players operation set $val math 40
scoreboard players operation $val math *= #5 math
scoreboard players operation $val math /= #100 math
```

Codice 10: Calcolo della percentuale con ordine di operazioni invertito

Nel primo caso, poiché $\frac{40}{100} = 0$ nel dominio degli interi, il risultato finale sarà 0: nella riga 3, infatti, viene eseguita l'operazione 0×5.
Nel secondo caso invece, si ottiene il risultato corretto pari a 2, poiché le operazioni vengono eseguite nell'ordine $40 \times 5 = 200$ e successivamente $\frac{200}{100} = 2$.

3.2. Assenza di Funzioni Matematiche

Poiché tramite *scoreboard* è possibile eseguire esclusivamente le quattro operazioni aritmetiche di base, il calcolo di funzioni più complesse quali logaritmi, esponenziali, radici quadrate o funzioni trigonometriche risulta particolarmente difficile da implementare.

Bisogna inoltre considerare il fatto che queste operazioni saranno ristrette al dominio dei numeri naturali. Si può dunque cercare un algoritmo che approssimi queste funzioni, oppure creare una *lookup table* [23].

```mcfunction
scoreboard players set #sign math -400
scoreboard players operation .in math %= #3600 const
execute if score .in math matches 1800.. run scoreboard players set #sign math 400
execute store result score #temp math run scoreboard players operation .in math %= #1800 const
scoreboard players remove #temp math 1800
execute store result score .out math run scoreboard players operation #temp math *= .in math
scoreboard players operation .out math *= #sign math
scoreboard players add #temp math 4050000
scoreboard players operation .out math /= #temp math
execute if score #sign math matches 400 run scoreboard players add .out math 1
```

Codice 11: Algoritmo che approssima la funzione $\sin(x)$.

La scrittura di algoritmi di questo tipo è impegnativa, e spesso richiede di gestire un input moltiplicato per 10^n il cui output è un intero dove sia assume che le ultime n cifre siano decimali[5]. Inoltre, questo approccio può facilmente provocare problemi di *integer overflow*.

Dunque, in seguito all'introduzione delle *macro*, si sono iniziate ad utilizzare le *lookup table*. Una *lookup table* consiste in un *array* salvato in uno *storage* che contiene tutti gli output di una funzione in un intervallo prefissato.

Ipotizziamo mi serva la radice quadrata con precisione decimale di tutti gli interi tra 0 e 100. Si può creare uno *storage* che contiene i valori $\sqrt{i}\ \forall i \in [0, 100] \cap \mathbb{N}$.

```mcfunction
data modify storage my_storage sqrt set value [
  0,
  1.0,
  1.4142135623730951,
  1.7320508075688772,
  2.0,
…                                   …
  10.0
]
```

Codice 12: *Lookup table* per $\sqrt{x}$, con $0 \leq x \leq 100$.

Dunque, data `get storage my_storage sqrt[4]` restituirà il quinto elemento dell'array, ovvero 2.0, l'equivalente di $\sqrt{4}$.

[5] Solitamente $n = 3$.

Dato che sono richiesti gli output di decine, se non centinaia di queste funzioni, i comandi per creare le *lookup table* vengono generati con script Python [24], ed eseguiti da *Minecraft* solamente quando si ricarica il *datapack*. Poiché queste strutture non sono soggette ad operazioni di scrittura, ma solo di lettura, non c'è il rischio che vengano modificate durante la sessione di gioco.

3.3. Alto Rischio di Conflitti

Nella sezione precedente è stato modificato lo *storage* `my_storage` per inserirvi un array. Si noti che non è stato specificato alcun *namespace*, per cui il sistema ha assegnato implicitamente quello predefinito, `minecraft:`.

Qualora un mondo contenesse due *datapack* sviluppati da autori diversi, ed entrambi modificassero `my_storage` senza indicare esplicitamente un *namespace*, potrebbero verificarsi conflitti.

Un'altra situazione che può portare a conflitti è quando due *datapack* sovrascrivono la stessa risorsa nel *namespace* `minecraft`. Se entrambi modificano `minecraft/loot_table/blocks/stone.json`, che determina gli oggetti si possono ottenere da un blocco di pietra, il compilatore utilizzerà il file del *datapack* che è stato caricato per ultimo.

Il rischio di sovrascrivere o utilizzare in modo improprio risorse appartenenti ad altri *datapack* non riguarda solo file che prevedono una *resource location*, ma si estende anche a componenti come *scoreboard* e *tag*.

In questo esempio sono presenti due *datapack*, sviluppati da autori diversi, con lo stesso obiettivo: eseguire una funzione sull'entità chiamante (`@s`) al termine di un determinato intervallo di tempo. In entrambi i casi, le funzioni incaricate dell'aggiornamento del timer vengono eseguite ogni *tick*, ovvero venti volte al secondo.

timer_a.mcfunction `mcfunction`

```
scoreboard players add @s timer 1
execute if score @s timer matches 20 run function some_function
```

timer_b.mcfunction `mcfunction`

```
scoreboard players remove @s timer 1
execute if score @s timer matches 0 run function some_function
```

Codice 13: Due funzioni che aggiornano un timer.

Le due funzioni modificano lo stesso *fakeplayer* all'interno dello stesso *scoreboard*. Poiché `timer_a` incrementa `timer` e `timer_b` lo decrementa, al termine di un *tick* il valore rimane invariato. Se invece entrambe variassero `timer` nello stesso verso, ad esempio incrementandolo, la durata effettiva del timer risulterebbe dimezzata. Questo è uno dei motivi per cui il nome di una *scoreboard* deve essere prefissato con un *namespace*, ad esempio `a.timer`[6].

Tra le varie condizioni per cui i selettori possono filtrare entità, ci sono i *tag*, ovvero stringhe memorizzate in un array nell'NBT di un entità.

Di conseguenza, se nell'esempio precedente gli sviluppatori necessitano che la funzione `timer` venga eseguita esclusivamente dalle entità contrassegnate da un determinato *tag*, ad esempio `has_timer`, i comandi per invocare `timer_a` e `timer_b` risulteranno i seguenti:

tick_a.mcfunction `mcfunction`

```
execute as @e[tag=has_timer] run function a:timer_a
```

tick_b.mcfunction `mcfunction`

```
execute as @e[tag=has_timer] run function b:timer_b
```

In entrambi i casi, `@e[tag=has_timer]` seleziona lo stesso insieme di entità. Ciò può risultare problematico se, allo scadere del timer di *b*, vengono eseguiti comandi che determinano comportamenti inaspettati o erronei per le entità del *datapack* di *a* (o viceversa).

Dunque, come per i nomi delle *scoreboard*, è buona norma prefissare i *tag* con il *namespace* del proprio progetto.

In conclusione, la convenzione vuole che si utilizzino prefissi anche per i nomi di *storage*, *scoreboard* e *tag*, nonostante i *datapack* compilino correttamente anche senza di essi.

3.4. Assenza di Code Blocks

Nei linguaggi di alto livello quali C o Java, i blocchi di codice che devono essere eseguiti condizionalmente o all'interno di un ciclo vengono racchiusi tra parentesi graffe. In Python, invece, la stessa funzione è ottenuta tramite l'indentazione del codice.

In una funzione *mcfunction*, questo costrutto non è supportato. Per eseguire una serie di comandi condizionalmente, è necessario creare un altro file che li contenga, oppure ripetere

[6] Come separatore si usa `.` e non `:` in quanto quest'ultimo è un carattere supportato nel nome di una *scoreboard*.

la stessa condizione su più righe. Quest'ultima opzione comporta maggiore *overhead*, specialmente quando il comando viene eseguito in più *tick*.

Di seguito viene riportato un esempio di come si può scrivere un blocco `if-else`, o `switch`, sfruttando il comando `return` per interrompere il flusso di esecuzione del codice nella funzione corrente.

```mcfunction
execute if entity @s[type=cow] run return run say I'm a cow
execute if entity @s[type=cat] run return run say I'm a cat
say I'm neither a cow or a cat
```

Codice 15: Funzione che in base all'entità esecutrice, stampa un messaggio diverso.

In questa funzione, i comandi dalla riga 2 in poi non verranno mai eseguiti se il tipo dell'entità è cow. Se la condizione alla riga 1 risulta falsa, l'esecuzione invece procede alla riga successiva, dove viene effettuato un nuovo controllo sul tipo dell'entità; anche in questo caso, se la condizione è soddisfatta, l'esecuzione si interrompe.

```
switch(entity){
  case "cow" -> print("I'm a cow")
  case "cat" -> print("I'm a cat")
  default -> print("I'm neither a cow or a cat")
}
```

Codice 16: Pseudocodice equivalente alla funzione precedente.

La funzione è abbastanza intuitiva, e corrisponde a qualcosa che si vedrebbe in un linguaggio di programmazione di alto livello. Ipotizziamo ora che si vogliano eseguire due o più comandi in base all'entità.

```mcfunction
execute if entity @s[type=cow] run return run say I'm a cow
execute if entity @s[type=cow] run return run say moo

execute if entity @s[type=cat] run return run say I'm a cat
execute if entity @s[type=cat] run return run say meow

say I'm neither a cow or a cat
```

Codice 17: Funzione errata per eseguire più comandi data una certa condizione.

Ora, se l'entità è di tipo `cow`, il comando alla riga 2 non verrà mai eseguito, anche se la condizione è soddisfatta. Dunque, è necessario creare una funzione che contenga quei due comandi.

main.mcfunction

```mcfunction
execute if entity@s[type=cow] run return run function is_cow
```

```
execute if entity@s[type=cat] run return run function is_cat

say I'm neither a cow or a cat
```

is_cow.mcfunction mcfunction

```
say I'm a cow
say moo
```

is_cat.mcfunction mcfunction

```
say I'm a cat
say meow
```

Considerando che i *datapack* si basano sull'esecuzione di funzioni in base a eventi già esistenti, sono numerosi i casi in cui ci si trova a creare più file che contengono un numero ridotto, purché significativo, di comandi.

Per quanto riguarda i cicli, come mostrato in Codice 7, l'unico modo per ripetere gli stessi comandi più volte è attraverso la ricorsione. Di conseguenza, ogni volta che è necessario implementare un ciclo, è indispensabile creare almeno una funzione dedicata. Se è invece necessario un contatore per tenere traccia dell'iterazione corrente (il classico indice `i` dei cicli `for`), è possibile utilizzare funzioni ricorsive che si richiamano passando come parametro una *macro*, il cui valore viene aggiornato all'interno del corpo della funzione. In alternativa, si possono scrivere esplicitamente i comandi necessari a gestire ciascun valore possibile, in modo analogo a quanto avviene con le *lookup table*.

Ipotizziamo si voglia determinare in quale *slot* dell'inventario del giocatore si trovi l'oggetto `diamond`. Una possibile soluzione è utilizzare una funzione che iteri da 0 a 35 (un giocatore può tenere fino a 36 oggetti diversi), dove il parametro della *macro* indica lo *slot* che si vuole controllare, ma questo approccio comporta un overhead maggiore rispetto alla verifica diretta, caso per caso, dei valori da 0 a 35.

find_diamond.mcfunction mcfunction

```
1    execute if items entity @s container.0 diamond run return run say slot 0
2    execute if items entity @s container.1 diamond run return run say slot 1
…                                        …
36   execute if items entity @s container.35 diamond run return run say slot
     35
```

In questa funzione, la ricerca viene interrotta da `return` appena si trova un diamante, ed è stato provato che abbia un *overhead* minore della ricorsione. Come nel caso delle *lookup table*, i file che fanno controlli di questo genere sono solitamente creati con script Python.

Infine, Codice 6 dimostra che, per utilizzare una *macro*, è sempre necessario creare una funzione capace di ricevere i parametri di un'altra funzione e applicarli a uno o più comandi indicati con `$`. Questa è probabilmente una delle ragioni più valide per cui sia richiesto scrivere una nuova funzione. Tuttavia, va comunque considerata tra file la cui creazione non è necessaria in un linguaggio di programmazione ad alto livello.

Dunque, programmando in *mcfunction* è necessario creare una funzione, ovvero un file, ogni volta che si necessita di:
- un blocco `if-else` che esegua più comandi;
- un ciclo;
- utilizzare una *macro*.

Ciò comporta un numero di file sproporzionato rispetto alle effettive righe di codice. Tuttavia, ci sono altre problematiche relative alla struttura delle cartelle e dei file nello sviluppo di *datapack* e *resourcepack*.

3.5. Organizzazione e Complessità della Struttura dei File

I problemi mostrati fin'ora sono prettamente legati alla sintassi dei comandi e ai limiti delle funzioni, tuttavia non sono da trascurare il quantitativo di file di un progetto.

Affinché *datapack* e *resourcepack* vengano riconosciuti dal compilatore, essi devono trovarsi rispettivamente nelle directory `.minecraft/saves/<world_name>/datapacks` e `.minecraft/resourcepacks`. Tuttavia, operare su queste due cartelle in modo separato può risultare oneroso, considerando l'elevato grado di interdipendenza tra i due sistemi. Lavorare direttamente dalla directory radice `.minecraft/` risulta poco pratico, poiché essa contiene un numero considerevole di file e cartelle non pertinenti allo sviluppo del *pack*.

Una possibile soluzione consiste nel creare una directory che contenga sia il *datapack* sia il *resourcepack* e, successivamente, utilizzare *symlink* o *junction* [25] per creare riferimenti dalle rispettive cartelle verso i percorsi in cui il compilatore si aspetta di trovarli.
I *symlink* (collegamenti simbolici) e le *junction* sono riferimenti a file o directory che consentono di accedere a un percorso diverso come se fosse locale, evitando la duplicazione dei contenuti.

Disporre di un'unica cartella radice contenente *datapack* e *resourcepack* semplifica notevolmente la gestione del progetto. In particolare, consente di creare una sola *repository* [26] Git [27], facilitando così il versionamento del codice, il tracciamento delle modifiche e la collaborazione tra più sviluppatori.

Attraverso il sistema delle *release* di GitHub [28] è possibile ottenere un link diretto a *datapack* e *resourcepack* pubblicati, che può poi essere utilizzato nei principali siti di hosting. Queste piattaforme, essendo spesso gestite da piccoli team di sviluppo, tendono ad affidarsi a servizi esterni per la memorizzazione dei file, come GitHub o altri provider.

Ipotizzando di operare in un ambiente di lavoro unificato, come quello illustrato in precedenza, viene presentato un esempio di struttura rappresentante i file necessari per introdurre un nuovo *item* [29] (oggetto). Sebbene l'*item* costituisca una delle funzionalità più semplici da implementare, la sua integrazione richiede comunque un numero non trascurabile di file.

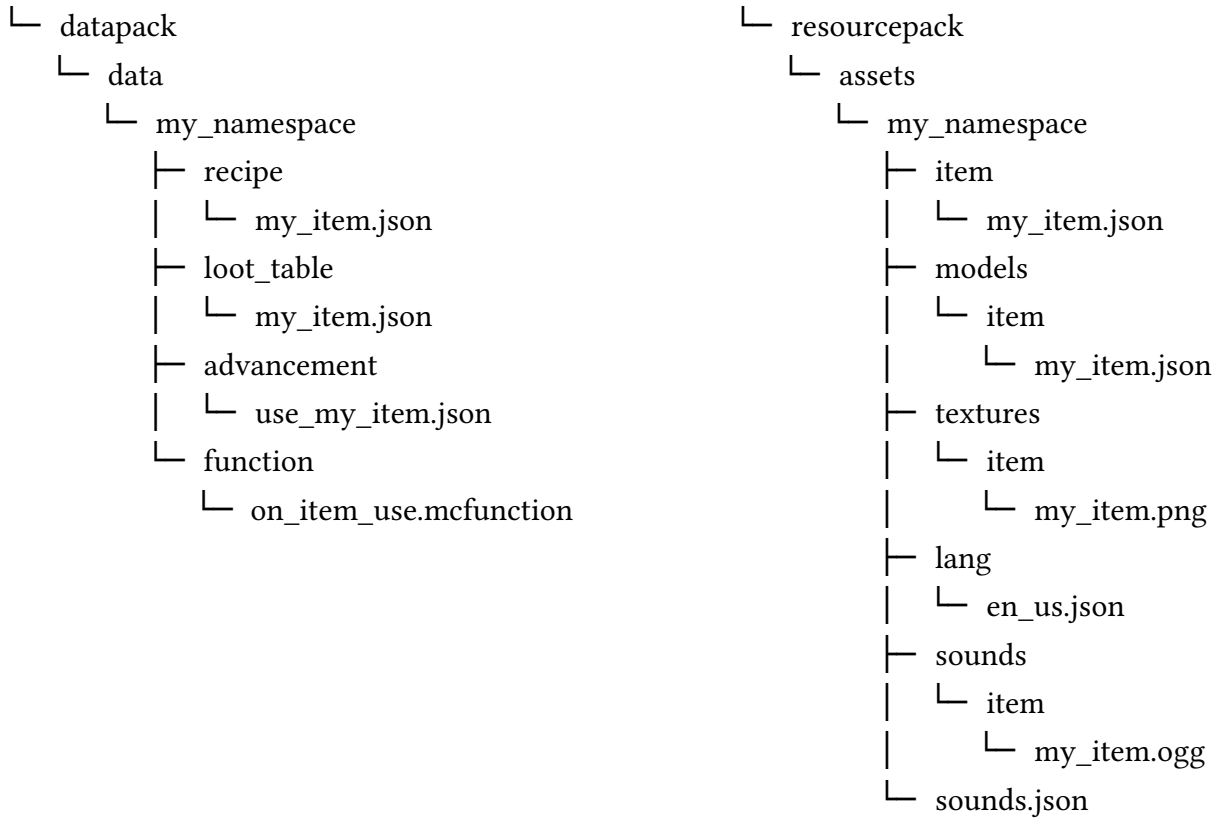

Figura 2: File necessari per implementare un semplice *item*.

Nella sezione *data*, che determina la logica e i contenuti, *loot_table* e *recipe* definiscono rispettivamente attributi dell'oggetto, e come questo può essere creato. L'*advancement* `use_my_item` serve a rilevare quando un giocatore usa l'oggetto, e chiama la funzione `on_item_use` che produrrà un suono.

I suoni devono essere collocati all'interno degli *assets*. Per poter essere riprodotti, ciascun suono deve avere un file audio in formato `.ogg` ed essere registrato nel file `sounds.json`. Nella cartella *lang* sono invece presenti i file responsabili della gestione delle traduzioni, organizzate come insiemi di coppie chiave-valore.
Per definire l'aspetto visivo dell'oggetto, si parte dalla sua *item model definition*, situata nella cartella `item`. Questa specifica il modello che l'*item* utilizzerà. Il modello 3D, collocato in `models/item`, ne definisce la forma geometrica, mentre la *texture* associata al modello è contenuta nella directory `textures/item`.

Si osserva quindi che, per implementare anche la *feature* più semplice, è necessario creare sette file e modificarne due. Pur riconoscendo che ciascun file svolge una funzione distinta e che la loro presenza è giustificata, risulterebbe certamente più comodo poter definire questo tipo di risorse *inline* [30].

Con il termine *inline* si intende la definizione e utilizzo una o più risorse direttamente all'interno dello stesso file in cui vengono impiegate. Questa modalità risulterebbe particolarmente vantaggiosa quando un file gestisce contenuti specifici e indipendenti. Ad esempio, nell'aggiunta di un nuovo item, il relativo modello e la *texture* non verrebbero mai condivisi con altri oggetti, rendendo superfluo separarli in file distinti.

Infine, l'elevato numero di file rende l'ambiente di lavoro complesso da navigare. In progetti di grossa portata questo implica, nel lungo periodo, una significativa quantità di tempo dedicata alla ricerca dei singoli file.

3.6. Stato dell'Arte delle Ottimizzazioni del Sistema

Alla luce delle numerose limitazioni di questo sistema, sono state rapidamente sviluppate soluzioni volte a rendere il processo di sviluppo più efficiente e intuitiva.

In primo luogo, gli stessi sviluppatori di *Minecraft* dispongono di strumenti interni che automatizzano la generazione dei file JSON necessari al corretto funzionamento di determinate *feature*. Durante lo sviluppo, tali file vengono creati automaticamente tramite codice Java eseguito in parallelo alla scrittura del codice sorgente, evitando così la necessità di definirli manualmente.

Un esempio lampante è il file `sounds.json`, che registra i suoni e definisce quali file `.ogg` utilizzare. Questo contiene quasi 25.000 righe di codice, ed è creato e aggiornato tramite software appositi ogni volta che viene inserita una *feature* che richiede un nuovo suono.

Tuttavia, questo software non è disponibile al pubblico, e anche se lo fosse, semplificherebbe la creazione solo dei file JSON, non di *mcfunction*. Dunque, sviluppatori indipendenti hanno realizzato dei propri precompilatori, progettati per generare automaticamente *datapack* e *resourcepack* tramite strumenti più intuitivi.

Un precompilatore è uno strumento che consente di scrivere le risorse e la logica di gioco in un linguaggio più semplice, astratto o strutturato, e di tradurle automaticamente nei numerosi file JSON, *mcfunction* e cartelle richieste dal gioco.
Il precompilatore al momento più completo e potente si chiama *beet* [31], e si basa sulla sintassi di Python, integrata con comandi di *Minecraft*.
Questo precompilatore, come molti altri, presenta due criticità principali:

- Elevata barriera d'ingresso: solo gli sviluppatori con una buona padronanza di Python sono in grado di sfruttarne appieno le potenzialità;
- Assenza di documentazione: la mancanza di una guida ufficiale rende il suo utilizzo accessibile quasi esclusivamente a chi è in grado di interpretare direttamente il codice sorgente di *beet*.

Altri precompilatori forniscono un'interfaccia più intuitiva e un utilizzo più immediato al costo di completezza delle funzionalità, limitandosi dunque a produrre solo una parte delle componenti che costituiscono l'ecosistema dei *pack*. Spesso, inoltre, la sintassi di questi linguaggi risulta più verbosa rispetto a quella dei comandi originali, poiché essi offrono esclusivamente un approccio programmatico alla composizione dei comandi senza portare ad alcun incremento nella loro velocità di scrittura.

```java
Execute myExecuteCommand = new Execute()
  .as("@a")
  .at("@s")
  .if("entity @s[tag=my_entity]")
  .run("say hello")
```

Questo risulta più articolato rispetto alla sintassi tradizionale `execute as @a at @s if entity @s[tag=my_entity] run say hello`.

Descrizione della Libreria

4.1. Approccio al Problema

Dato il contesto descritto e le limitazioni degli strumenti esistenti, ho cercato una soluzione che permettesse di ridurre la complessità d'uso senza sacrificare la completezza delle funzionalità. Di seguito verranno illustrate le principali decisioni progettuali e le ragioni che hanno portato alla scelta del linguaggio di sviluppo.

Inizialmente ho tentato di progettare un *superset* [32] di *mcfunction*, ossia un linguaggio che estende quello originale introducendo nuove funzionalità mantenendone allo stesso tempo la compatibilità. Questo linguaggio avrebbe consentito di dichiarare e utilizzare più elementi (*mcfunction* e JSON), all'interno di un unico file, arricchendo anche la sintassi con elementi di zucchero sintattico volti a semplificare la scrittura delle parti più verbose.

```mcf
package foo
scoreboard players operation @s var *= 4
if score @s var matches 10.. run function {
  say hello
  say something else
}
```

Codice 20: Esempio di questo *superset*, caratterizzato da file con l'estensione `.mcf`

Eseguendo questo codice, non solo si sarebbe creata la funzione dichiarata all'interno delle parentesi graffe, ma inserito il namespace prima di `var`, e creato il comando che assegna alla costante `#4` i valore 4. Come è stato mostrato nel Codice 8, per eseguire divisioni e moltiplicazioni per valori costanti, è prima necessario definirli in uno *score*. Compilando il frammento di codice dell'esempio, si sarebbero ottenuti i seguenti file:

load.mcfunction `mcfunction`
```
scoreboard players set #4 foo.var 4
```

main.mcfunction `mcfunction`
```
scoreboard players operation @s foo.var *= #4 foo.var
execute if score @s foo.var matches 10.. run function foo:5a3c50
```

5a3c50.mcfunction `mcfunction`
```
say hello
say something else
```

Ho inizialmente scelto di utilizzare la versione Java della libreria ANTLR [33] per definire la grammatica del linguaggio. Tuttavia, mi sono presto reso conto che realizzare una grammatica in grado di cogliere tutte le sfumature della sintassi di *mcfunction*, integrandovi al contempo le mie estensioni, avrebbe richiesto un impegno di sviluppo superiore a quello compatibile con un progetto di tirocinio.

Ho quindi pensato di sviluppare una libreria che permetta di definire la struttura di un *pack*, dalla radice del progetto fino ai singoli file, sotto forma di oggetti, affinché sia possibile rappresentare l'intero insieme delle risorse come una struttura dati ad albero n-ario. Questa, al momento dell'esecuzione, è attraversata per generare automaticamente i file e le cartelle corrispondenti ai nodi, all'interno delle directory di *datapack* e *resourcepack*.

Il principale vantaggio di questo approccio consiste nella possibilità di definire più nodi all'interno dello stesso file, evitando così la frammentazione del codice e semplificando la gestione della struttura complessiva del *pack*. Inoltre, l'impiego di un linguaggio ad alto livello consente di sfruttare costrutti quali cicli e funzioni per automatizzare la generazione di comandi ripetitivi (ad esempio le già citate *lookup table*). La rappresentazione a oggetti della struttura

permette anche di definire metodi di utilità per accedere e modificare i nodi da qualsiasi punto del progetto. Ad esempio, si può implementare un metodo `addTranslation(key, value)` che permette di aggiungere, indipendentemente dal contesto in cui viene invocato, una nuova voce nel file delle traduzioni.

Dunque ho pensato a quale linguaggio di programmazione tra Python e Java si potesse usare per realizzare questa libreria.

	Vantaggi	Svantaggi
Python	• Gestione semplice di stringhe (`f-string` [34]) e file JSON; • Sintassi concisa; • Facilmente distribuibile.	• Non nativamente orientato agli oggetti; • Tipizzazione dinamica che può causare errori a runtime; • Prestazioni inferiori in fase di esecuzione.
Java	• Maggiore familiarità con progetti di grandi dimensioni; • Completamente orientato agli oggetti; • Compilazione ed esecuzione più efficienti.	• Assenza di `f-strings` e manipolazione delle stringhe più complessa; • Gestione dei file JSON più verbosa; • Sintassi più prolissa, che rallenta la scrittura del codice.

Tabella 1: Java e Python a confronto.

Dopo un'attenta analisi, ho scelto di utilizzare Java per lo sviluppo del progetto, poiché secondo me è lo strumento ideale per applicare *design pattern* in grado di semplificare e rendere più robusta l'implementazione, anche a costo di sacrificare parzialmente la comodità d'uso per l'utente finale.
Inoltre, il tipaggio statico di Java permette di identificare in fase di sviluppo eventuali utilizzi impropri di oggetti o metodi della libreria, consentendo anche agli utenti meno esperti di comprendere più facilmente il funzionamento del sistema.

Il progetto, denominato *Object Oriented Pack* (OOPACK), è organizzato in 4 sezioni principali.

`internal` Contiene classi astratte e interfacce che riproducono la struttura di un generico *filesystem*. Classi e metodi di questo *package* [35] non saranno mai utilizzate dal programmatore.

`objects` Contiene le classi che rappresentano gli oggetti utilizzati nei *datapack* e *resourcepack*.

`util` Raccoglie metodi di utilità impiegati sia per il funzionamento del progetto, sia a supporto del programmatore (ponendo attenzione alla visibilità dei singoli metodi).

Radice del progetto Contiene gli oggetti principali che descrivono struttura di un *pack* (`Datapack`, `Resourcepack`, `Namespace`, `Project`).

4.2. Classi Astratte e Interfacce

4.2.1. Buildable

L'obiettivo della libreria sviluppata è delegare la creazione dei file che compongono un *pack* al metodo `build()`, definito nella classe di più alto livello, `Project`. Di conseguenza, ogni oggetto appartenente al progetto deve essere *buildable*, ovvero «costruibile», in modo da poter generare il corrispondente file. L'interfaccia `Buildable` definisce il contratto che stabilisce quali oggetti possono essere costruiti attraverso il metodo `build()`.

```java
public interface Buildable {
    void build(Path parent);
}
```

Il parametro `parent` rappresenta un oggetto di tipo `Path` [36] che indica la directory di destinazione nella memoria locale in cui verrà scritto il file. Durante il processo di costruzione del progetto, questo percorso viene progressivamente esteso aggiungendo sottocartelle, fino a individuare la posizione finale del file generato.

L'interfaccia `FileSystemObject` estende `Buildable` con lo scopo di rappresentare file e cartelle del *file system*. Definisce il contratto `getContent()`, che specifica il contenuto associato all'oggetto. In base al tipo di classe che lo implementa, potrà restituire un qualche tipo di dato (file) o una lista di `FileSystemObject` (cartella).

Questa interfaccia definisce il metodo statico `find()` usato per trovare un `file` all'interno di un `FileSystemObject` che soddisfa una certa condizione.

```java
static <T extends FileSystemObject> Optional<T> find(
            FileSystemObject root,
            Class<T> clazz,
            Predicate<T> condition
    ) {
        if (clazz.isInstance(root)) {
            T casted = clazz.cast(root);
            if (condition.test(casted)) {
                return Optional.of(casted);
            }
        }
        Object content = root.getContent();
        if (content instanceof Set<?> children) {
            for (Object child : children) {
                Optional<T> found = find((FileSystemObject) child, clazz,
                condition);
                if (found.isPresent()) {
                        return found;
                }
            }
        }
        return Optional.empty();
    }
```

Questo metodo generico prende in input un `FileSystemObject` (non sa se si tratta di una cartella o file), la classe del tipo ricercato (`clazz`), e una condizione da soddisfare affinché l'oggetto risulti trovato. Esegue i seguenti passi:

1. controlla se il nodo attuale è un istanza del tipo cercato;
2. in caso positivo:
   1. applica la condizione passata come `Predicate` [37];
   2. se è soddisfatta, l'oggetto è trovato e viene restituito un `Optional` [38] contenente l'oggetto.
3. in caso negativo, continua la ricerca nei figli;
4. ottiene il contenuto del nodo corrente;
5. se il contenuto è un `Set` [39] (dunque il nodo è una cartella):
   1. richiama `find(...)` su ciascun elemento figlio;
   2. se uno dei figli contiene l'oggetto cercato, interrompe la ricerca.
6. altrimenti restituisce un `Optional` vuoto, indicando che l'elemento non è stato trovato.

`FileSystemObject` definisce anche il contratto `collectByType(Namespace data, Namespace assets)`. Questo sarà sovrascritto per indicare se l'oggetto appartiene alla categoria *data* dei *datapack* o *assets* dei *resourcepack*.

4.2.2. AbstractFile e AbstractFolder

Tutti gli oggetti rappresentati file nel progetto, che saranno successivamente scritti in memoria, sono un estensione della classe `AbstractFile`.
`AbstractFile<T>` è una classe astratta parametrizzata con un tipo generico `T`, che rappresenta il contenuto del file, memorizzato nell'attributo `content`. La classe dispone dell'attributo `name`, che specifica il nome del file da creare, privo di estensione. Possiede inoltre un riferimento al `parent`, ovvero alla sottocartella o cartella delle risorse in cui il file si troverà. L'oggetto dispone infine di un riferimento al *namespace* in cui si trova.
`namespace` è formattato per comporre assieme `name` la stringa che corrisponde alla *resource location* dell'oggetto corrente. Questa logica è implementata nel metodo `toString()`, così che l'istanza possa essere inserita direttamente in altre stringhe restituendo automaticamente il riferimento completo alla risorsa.

```java
@Override
public String toString() {
  return String.format("%s:%s", getNamespaceId(), getName());
}
```

`AbstractFile`, oltre ad implementare `FileSystemObject`, implementa `PackFolder` ed `Extension`.
`PackFolder` fornisce un solo contratto, `getFolderName()` che definisce il nome della cartella in cui sarà collocato. Ad esempio l'oggetto `Function` eseguirà l'*override* di questo metodo per restituire `"function"`, dal momento che tutte le funzioni devono essere nella cartella `function`.
Similmente, l'interfaccia `Extension`, tramite il contratto `getExtension()` permetterà agli oggetti che estendono `AbstractFile` di indicare la propria estensione (`.json`, `.mcfunction`).

L'altra classe astratta che implementa `FileSystemObject` è `AbstractFolder`. Questa classe astratta parametrizzata con `<T extends FileSystemObject>` dispone di un attributo `children` di tipo `Set<T>`, usato per mantenere riferimenti a nodi che estendono esclusivamente `FileSystemObject`, evitando duplicati. Il suo metodo `build()` invoca a sua volta `build()` per ogni figlio.
Il metodo `collectByType(...)` esegue invece una chiamata polimorfica a `collectByType` su ogni nodo figlio, propagando la divisione di oggetti attraverso l'intera struttura ad albero.

4.2.3. Folder e ContextItem

La classe `Folder` estende `AbstractFolder<FileSystemObject>`. I suoi `children` saranno dunque `FileSystemObject`. Dispone di un metodo `add()` per aggiungere un elemento all'insieme dei figli. Questo viene usato dalla logica interna della liberia, ma non è pensato per l'utilizzo dell'utente finale.

Nella fase iniziale di sviluppo del progetto, la creazione di una cartella con dei figli richiedeva l'istanza di un oggetto `Folder` e la successiva invocazione del metodo `add(...)`, passando come parametro uno o più oggetti generati manualmente tramite l'operatore `new`.
Un sistema basato sulla creazione diretta degli oggetti presenta diverse limitazioni. In primo luogo, introduce un forte accoppiamento tra il codice *client* e le classi concrete: qualsiasi modifica ai costruttori richiederebbe di aggiornare manualmente ogni punto del codice in cui tali oggetti vengono istanziati. Inoltre, l'utilizzo di espressioni come `myFolder.add(new Function(...))` risulta poco pratico per l'utente finale, soprattutto se l'obiettivo è offrire un'interfaccia più semplice e immediata per la creazione dei file.

Dunque, ho modificato il sistema per appoggiarsi su un oggetto `Context` che indica il *parent*, ovvero la cartella in cui si sta lavorando. La classe `Context` contiene un attributo statico e privato di tipo `Stack<ContextItem>` [40]. Questo è usato per tenere traccia del livello di *nesting* delle cartelle. `stack.peek()` restituisce il `ContextItem` in cima allo `stack`, ovvero quello in cui si sta lavorando al momento.

L'interfaccia `ContextItem` fornisce il metodo `add()` che un qualsiasi contenitore di oggetti implementerà (non solo `Folder`, ma come si vedrà successivamente, anche `Namespace` in quanto anche esso è contenitore di `FileSystemObject`).
L'interfaccia dispone anche di due metodi `default` per indicare quando si vuole operare nel contesto relativo a quell'oggetto.

```java
default void enter() {
    Context.enter(this);
  }
default void exit() {
    Context.exit();
  }
```

Codice 25: Metodi dell'interfaccia `ContextItem`.

Invocando `enter()`, si sta aggiungendo l'oggetto che implementa `ContextItem` in cima allo `stack` del contesto, indicando che è la cartella in cui verranno aggiunti tutti i prossimi `FileSystemObject`. Per rimuovere l'oggetto dalla cima dello `stack`, si chiama il metodo `exit()`.
Con questo sistema, il programmatore può spostarsi tra diversi livelli della struttura del *filesystem* in modo rapido e controllato, senza dover passare manualmente riferimenti ai vari contenitori.

4.2.4. Utilizzo delle Factory

Come fa un oggetto che estende `FileSystemObject` a sapere in quale `ContextItem` deve essere inserito? Per gestire automaticamente questo aspetto e al tempo stesso evitare la creazione diretta tramite `new`, si ricorre al design pattern *factory*.

Le *factory* sono un modello di progettazione che ha lo scopo di separare la logica di creazione degli oggetti dal codice che li utilizza. Invece di istanziare le classi direttamente, il client si limita a chiedere alla *factory* di creare l'oggetto desiderato. Sarà la *factory* a occuparsi di scegliere quale classe concreta istanziare e con che stato. Nel nostro caso, si occuperà anche di inserirla nel contesto in cima allo `stack`.

Un'evoluzione di questo concetto è l'*abstract factory*, un pattern che fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti tra loro, senza specificare le loro classi concrete.
L'*abstract factory* non crea direttamente gli oggetti, ma definisce un insieme di metodi di creazione che le sottoclassi concrete implementano per produrre versioni specifiche di tali oggetti.

Questo risulta particolarmente utile nel nostro contesto, in quanto si vuole dare all'utente la possibilità di istanziare oggetti in modi diversi.

```java
public interface FileFactory<F> {
    F ofName(String name, String content, Object... args);
    F of(String content, Object... args);
}
```

Codice 26: Interfaccia `FileFactory`.

L'utente può specificare manualmente il nome del file da costruire, oppure lasciare che sia la libreria a generare un nome casuale. Se il nome contiene uno o più `/`, verranno letti come cartelle.
Il nome assegnato all'oggetto non influisce sul funzionamento della libreria, dal momento che, quando l'oggetto viene utilizzato in un contesto testuale, la chiamata implicita al metodo `toString()` restituisca il riferimento alla sua *resource location*.
Gli oggetti passati come parametro *variable arguments* [41] (*varargs*, `Object... args`) sostituiranno i corrispondenti valori segnaposto (`%s`), interpolando così il contenuto testuale prima che il file venga scritto su disco.

4.2.5. Classi File Astratte

L'interfaccia `FileFactory` è implementata come classe annidata all'interno dell'oggetto astratto `PlainFile`, il quale rappresenta qualsiasi tipo di file che non contiene suoni o immagini (ovvero file di testo o dati generici).
Questa *nested class*, chiamata `Factory`, dispone di due parametri e serve a istanziare le sottoclassi di `PlainFile`.

```java
protected static class Factory<
  F extends PlainFile<C>,
  C
> implements FileFactory<F>
```

Codice 27: Intestazione della classe `Factory` per `PlainFile`

`F` è un tipo generico che estende `PlainFile<C>` e rappresenta il tipo di file che la classe istanzierà. Vincolando `F` a `PlainFile<C>`, la *factory* garantisce che tutti i file creati abbiano un contenuto di tipo `C` e siano sottoclassi di `PlainFile`.
Il contenuto `C` del file è dettato dalle sottoclassi che ereditano `PlainFile`. Questo permette alla *factory* di essere generica, creando file con contenuti diversi senza riscrivere codice.

La *factory* possiede un riferimento all'oggetto `Class` [42], parametrizzato con il tipo `F`, degli oggetti che istanzierà ed è utilizzato nel metodo `instantiate()`. Questo restituisce l'oggetto da creare dati due parametri: il nome del file da creare, e il suo contenuto (di tipo `Object`, dato che ancora si sta operando in un contesto generico). La funzione esegue le seguenti istruzioni per istanziare l'oggetto:

1. ottiene un riferimento alla classe del contenuto (`StringBuilder.class` o `JsonObject.class`). Questo è usato per individuare il costruttore della classe `F`;
2. recupera il costruttore tramite *reflection*. Controlla che la classe `F` abbia un costruttore che disponga dei seguenti parametri: `String name` e `C content`;
3. rende accessibile il costruttore. Senza questo passo, non sarebbe possibile accedere ai costruttori privati o protetti;
4. crea un'istanza della classe;
5. aggiunge l'istanza al contesto attuale;
6. restituisce l'oggetto creato.

`AbstractFile` è esteso da `TextFile`, il cui `content` è di tipo `StringBuilder` [43], e `JsonFile`, che utilizza invece `JsonObject` [44] come contenuto.

`TextFile` rappresenta un file di testo generico, il cui contenuto è gestito tramite un oggetto `StringBuilder`, così da consentire operazioni di concatenazione delle stringhe in modo efficiente. L'unica classe che la estende è `Function`, poiché è l'unico tipo di file nel progetto che prevede la scrittura diretta di testo.

`JsonFile` è invece la classe base ereditata da tutti gli altri file di un *pack*. Il suo contenuto è di tipo `JsonObject`, affinché si possano gestire e manipolare facilmente dati in formato JSON tramite la libreria *GSON* [45] di Google.
La *factory* di `JsonFile` eredita quella di `PlainFile`, aggiungendovi metodi.

```java
protected static class Factory<F extends JsonFile>
  extends PlainFile.Factory<F, JsonObject>
  implements JsonFileFactory<F>
```

Codice 28: Intestazione della classe `Factory` per `JsonFile`.

L'estratto di codice riportato definisce la *factory* incaricata di istanziare esclusivamente classi che estendono `JsonFile`. Questa classe eredita la factory di `PlainFile`, specializzandola per gestire contenuti di tipo `JsonObject`. Inoltre, implementa l'interfaccia `JsonFileFactory`, la quale definisce i metodi di creazione specifici per i file JSON, che dunque hanno come parametro `JsonObject`.
Nella classe `JsonFile` viene anche eseguito l'*override* del metodo `getExtension()` per restituire la stringa `"json"`.

I costruttori delle classi sopra descritte richiedono un contenuto di tipo diverso da `String`. In entrambi i casi viene fatto un leggero *parsing* prima della scrittura sul file. Oltre alla già citata sostituzione di valori segnaposto, dopo che `StringBuilder` e `JsonObject` sono stati convertiti in stringhe, si controlla il contenuto per alcuni pattern.
La sottostringa `"$ns$"` verrà sostituita con il nome effettivo del *namespace* attivo al momento della costruzione, mentre `"$name$"` verrà sostituito con la propria *resource location*.
Quest'ultimo risulta particolarmente utile nei casi di dipendenze circolari, in cui può essere richiesto il nome di un oggetto prima che esso sia effettivamente istanziato, dal momento che non è ancora possibile ottenere la sua rappresentazione testuale tramite *casting* implicito a stringa.

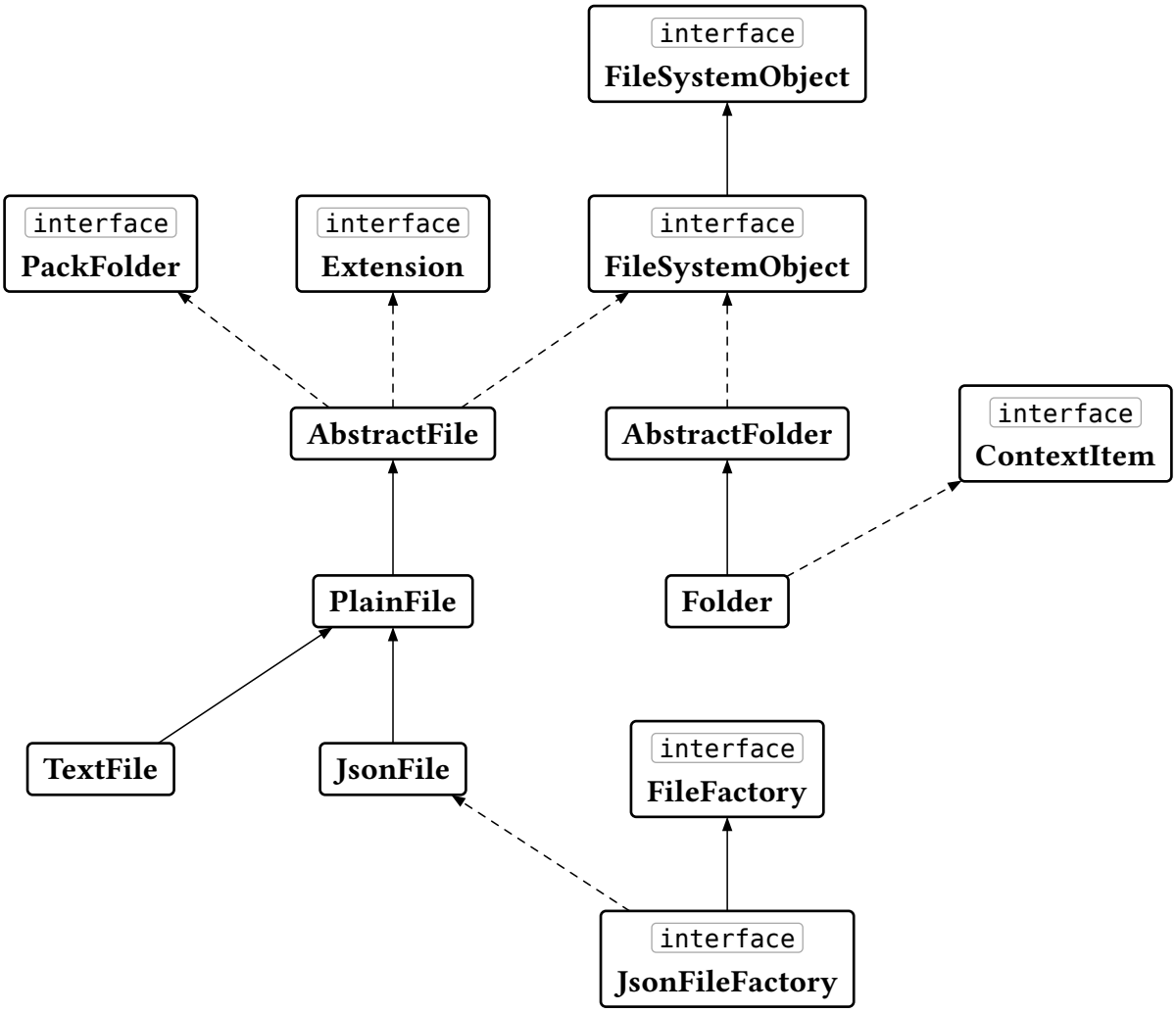

Figura 3: Diagramma del sistema progettato fino a questo punto.

Nella struttura riportata non sono ancora stati definiti metodi o classi specifiche per l'implementazione di un *pack*. Ritengo che questo livello di astrazione sia potenzialmente applicabile anche in altri contesti, in quanto permette di generare in modo sistematico più file a partire da un'unica definizione di riferimento. Questo approccio potrebbe risultare particolarmente utile anche in altre DSL caratterizzate da vincoli strutturali, dove la generazione automatizzata di file correlati è un requisito per la scalabilità e la manutenibilità del codice.
Di seguito invece si esporranno elementi e funzionalità definite appositamente per lo sviluppo dei *pack*.

4.3. Classi Concrete

4.3.1. File Concreti e Module

Le classi astratte `DataJson` e `AssetsJson` sono sottoclassi di `JsonFile`, e hanno il compito di eseguire un *override* del metodo `collectByType()` di `FileSystemObject` per indicare se il file che rappresentano appartiene alla categoria *datapack* o *resourcepack*.

```java
@Override
 public void collectByType(Namespace data,Namespace assets) {
     data.add(this);
 }
```

Codice 29: metodo `collectByType()` di `DataJson`.

Queste classi saranno poi ereditate dalle classi concrete dei file che compongono un *pack*.

Unica eccezione è la classe `Function`. Questa estende `TextFile`, indicando la propria estensione (`.mcfunction`) con *override* del metodo `getExtension()`, e anche il proprio tipo come visto nell'esempio sopra con `DataJson`. Dato che `TextFile` non dispone di una *factory* per file di testo non in formato JSON, sarà la *factory* di `Function` stessa a estendere `PlainFile.Factory`, definendo come parametro per il contenuto del file `StringBuilder`, e come oggetto istanziato `Function`.

Le classi rappresentanti file di alto livello sono dotate di attributo statico e pubblico di tipo `JsonFileFactory<...>`, parametrizzato per la classe specifica che istanzia. Queste classi sono 39 in totale, e ognuna corrisponde a un specifico oggetto utile al funzionamento di un *datapack* o *resourcepack* (30 e 9 rispettivamente). Dal momento che ognuna di queste classi deve disporre di una *factory*, un costruttore, e dell'*override* al metodo `getFolderName()`, ho scelto di usare una libreria per generare il loro codice Java.

Un'alternativa possibile sarebbe potuta consistere nel definire un metodo statico generico all'interno di `JsonFile.Factory`, che richiede come parametri il tipo della classe da istanziare e la cartella corrispondente. Così facendo non sarebbe necessario creare una classe dedicata per ciascun tipo di file, ma risulterebbe sufficiente invocare direttamente la funzione `create()` per generare l'istanza desiderata.

```java
Advancement adv = JsonFile.Factory.create(
  Advancement.class,
  "advancement",
  json
);
Model model = JsonFile.Factory.create(Model.class, "model", json);
```

Codice 30: Esempio di approccio alternativo.

Tuttavia è evidente che non risulta comodo per l'utente finale dover specificare tutti questi parametri ogni volta che si vuole usare la *factory*.
Dunque ho scritto una classe di utilità `CodeGen` che sfrutta la libreria *JavaPoet* [46] per creare le classi e i metodi al loro interno. In questo modo per creare un modello si può semplicemente scrivere `Model.f.of(json)`.

Sono disponibili anche classi rappresentanti file binari. Queste non ereditano la `Factory` di `PlainFile`, ma usano *factory* proprie per istanziare `Texture` e `Sound`.

L'oggetto `Texture` estende un `AbstractFile` che ha come contenuto una `BufferedImage` [47]. Se viene passata una stringa al suo metodo `of()`, verrà convertita in un path che punta alla cartella `resources/texture` del progetto Java. Si può anche passare direttamente una `BufferedImage`, creata dinamicamente tramite codice Java.

I suoni invece usano come contenuto un array di byte. La loro *factory*, similmente a quella di `Texture`, permette di caricare suoni dalle risorse del progetto (`resources/sound`).

Ho voluto creare una sottoclasse astratta di `Folder`, chiamata `Module`, con lo scopo di invitare ulteriormente a scrivere codice «modulare», dove c'è una chiara divisione di compiti e raggruppamento di contenuti affini. Ad esempio, se sto implementando una feature A, tutte le risorse e dati relative ad A, potranno essere inserite nel `Module` A.

La classe dispone di un *entry point*, ovvero una funzione astratta `content()` che verrà sovrascritta da tutte le classi che erediteranno `Module`, con lo scopo di fornire un chiaro punto in cui definire la logica interna del modulo.

I moduli vengono istanziati tramite il metodo `register(Class<? extends Module>... classes)`, che invoca il costruttore di una o più classi che estendono `Module`.

Quando un nuovo modulo viene istanziato, il costruttore imposta la nuova istanza come contesto corrente. Successivamente viene invocato il metodo `content()`, tramite il quale viene eseguito il codice specifico del modulo. Al termine di questa esecuzione, il costruttore ripristina il contesto precedente chiamando il metodo `exit()` dei `ContextItem`. In questo modo si garantisce che l'esecuzione di ciascun modulo avvenga in maniera indipendente, evitando che compili in un contesto non pertinente.

4.3.2. Namespace, Project

Le classi concrete vengono raccolte da `Namespace`. Come i `Folder`, dispongono di un `Set` che contiene i figli, ed implementa le interfacce `Buildable` e `ContextItem`. Quest'ultima viene utilizzata perché un `Project` può essere composto da più *namespace*, quindi bisogna tenere traccia di quello corrente in cui si aggiungono i `FileSystemObject` appena creati.
I *children* di `Namespace` possono essere di natura *data* o *assets*, dunque prima che vengano scritti su file sarà necessario dividerli nelle cartelle corrispondenti.

La classe presenta una particolarità nel suo metodo `exit()`, usato per dichiarare quando non si vogliono più creare file su questo *namespace*. Oltre a indicare all'oggetto `Context` di chiamare `pop()` sul suo `stack` interno, viene anche chiamato il metodo `addNamespace()` di `Project` che verrà mostrato in seguito.

La classe `Project` rappresenta la radice del progetto che verrà creato, e contiene informazioni essenziali per l'esportazione del progetto. Queste verranno impostate dall'utente finale tramite un *builder*.

Il *builder pattern* è un *design pattern* creazionale utilizzato per costruire oggetti complessi progressivamente, separando la logica di costruzione da quella di istanziazione dell'oggetto. È particolarmente utile quando un oggetto ha molti parametri opzionali, come nel caso di `Project`.
Tramite la classe `Builder` di `Project`, si possono specificare:

- nome del mondo, ovvero in quale *save file* verrà esportato il *datapack*
- nome del progetto;
- versione del *pack*. Questa verrà usata per comporre il nome delle cartelle *datapack* e *resourcepack* esportate, e anche per ottenere il loro rispettivo `pack_format` richiesto;
- *path* dell'icona di *datapack* e *resourcepack*, che verrà prelevata dalle risorse;
- descrizione in formato JSON o stringa di *datapack* e *resourcepack*, richiesta dal file `pack.mcmeta` di entrambi.
- uno o più *build path*, ovvero la cartella radice in cui verrà esportato l'intero progetto. In genere questa coinciderà con la cartella globale di minecraft, nella quale sono raccolti tutti i *resourcepack* e i *save file*, tra cui quello in cui si vuole esportare il *datapack*.

Dopo aver definito questi valori, il progetto sarà in grado di identificare il *path* cui dovrà esportare le cartelle radice di *datapack* e *resourcepack*.

Un altro *design pattern* creazionale applicato a `Project` è *singleton*, il cui scopo è garantire che una classe abbia una sola istanza in tutto il programma e che sia facilmente accessibile da qualunque punto del codice. Questo viene implementato tramite una variabile statica e privata di tipo `Project` all'interno della classe stessa. Un riferimento ad essa è ottenuto con il metodo `getInstance()`, che solleva un errore nel caso il progetto non sia ancora stato costruito con il `Builder`.

`Project` dispone al suo interno di attributi di tipo `Datapack` e `Resourcepack`. Questi hanno il compito di contenere i file che saranno scritti su memoria rigida ed estendono la classe astratta `GenericPack`.
`GenericPack` implementa le interfacce `Buildable` e `Versionable`. Quest'ultima fornisce i metodi per ottenere i *pack format* corrispettivi alla versione del progetto.
Fornisce inoltre l'attributo `namespaces` di tipo `Map` [48], nel quale verranno salvati i corrispettivi `Namespace`. Tramite il suo metodo `makeMcMeta()` viene generata la struttura JSON che specifica il format (*minor* e *major*) e la descrizione della cartella.

Il metodo `build()`, è sovrascritto per farlo iterare su tutti i valori del dizionario `namespaces`, affinché anch'essi vengano costruiti.

Il metodo `addNamespace()`, accennato precedentemente, non aggiunge direttamente il *namespace* al progetto. Prima divide i `FileSystemObject` contenuti in quelli inerenti alle risorse (*assets*) e quelli relativi alla logica (*data*). Questa suddivisione viene fatta chiamando il metodo precedentemente citato `collectByType()`. Al termine della divisione si avranno due nuovi *namespace* omonimi, ma con i contenuti divisi per funzionalità. Il *namespace* che contiene i file di *data* sarà aggiunto alla lista di `Namespace` di `datapack`. Se il *namespace* contenente gli *assets* non è vuoto, verrà aggiunto a quelli di `resourcepack`.

Quindi chiamate al metodo build si propagheranno inizialmente da `Project`, poi ai suoi campi `datapack` e `resourcepack`, questi la invocheranno sui loro `namespace`. Questi a loro volta lo invocheranno su tutti i loro figli (cartelle e file), ricoprendo così l'intero albero.

Con gli oggetti descritti fin'ora è possibile costruire un *pack* a partire da codice Java, tuttavia si possono sfruttare ulteriormente proprietà del linguaggio di programmazione per implementare funzioni di utilità, che semplificano ulteriormente lo sviluppo.

4.4. Utilità

4.4.1. Trova o Crea File

Il metodo `find()`, descritto precedentemente (Codice 23), è impiegato in metodi di utilità che permettono di modificare i contenuti di file, in particolare quelli soggetti a modifiche da più punti del codice. Ad esempio i file `lang`, che contengono le traduzioni, devono essere continuamente aggiornati con nuove voci. Similmente, ogni nuovo suono deve essere registrato nel file `sounds.json`. Come accennato in precedenza, quando questi file di risorse vengono utilizzati dagli sviluppatori di *Minecraft*, non vengono compilati manualmente, ma generati automaticamente tramite codice Java proprietario.

Poiché questi file non sono stati concepiti per essere modificati manualmente, ho deciso di implementare nella classe `Util` metodi dedicati per aggiungere elementi alle risorse in modo programmatico, accessibili da qualunque parte del progetto.
Ho prima scritto una funzione che permette di ottenere un riferimento all'oggetto ricercato, o di crearne uno nuovo qualora non fosse trovato.

```java
private static <T extends JsonFile> T getOrCreateJsonFile(
        Namespace namespace,
        Class<T> clazz,
        String name,
        Supplier<T> creator
) {
    return namespace.getContent().stream()
            .map(child -> FileSystemObject.find(child,
                    clazz,
                    file -> file.getName().equals(name)))
            .filter(Optional::isPresent)
            .map(Optional::get)
            .findFirst()
            .orElseGet(creator);
}
```

Codice 31: Metodo che sfrutta la programmazione funzionale per restituire il `JsonFile` cercato.

Il metodo richiede la classe del tipo che si sta cercando, il suo nome e un `Supplier` [49]. Esegue i seguenti passi:

1. ottiene l'insieme dei figli del `namespace` in cui effettuare la ricerca, e ne crea uno `Stream` [50] per l'elaborazione funzionale;
2. ogni `child` è trasformato in un `Optional`:
   1. per ogni `child` dello `stream`, invoca il metodo `find()`, specificando la classe e una condizione che determina il successo della ricerca (`Predicate`);
   2. `find()` restituisce un `Optional`. Questo sarà vuoto se la ricerca non ha avuto successo;
3. scarta gli `Optional` vuoti;
4. estrae i valori degli `Optional` rimasti;
5. ottiene un `Optional` contenente il primo elemento trovato (`findFirst()`). Se non è presente alcun elemento, restituisce un `Optional` vuoto;
6. se l'`Optional` è vuoto, il `Supplier` fornisce una nuova istanza dell'oggetto da restituire.

In questo modo si garantisce che il metodo restituisca sempre o l'oggetto ricercato, oppure ne viene istanziato uno nuovo. Il metodo `orElseGet()` di Java rappresenta un'applicazione del *design pattern lazy loading*, che differisce dal tradizionale `orElse()` per l'uso di un `Supplier` che viene invocato solo se l'`Optional` è vuoto. Questo approccio consente di ritardare la creazione di un oggetto fino al momento in cui è effettivamente necessario, rendendo il sistema leggermente più efficiente in termini di memoria [51], [52].

La funzione appena mostrata è applicata in numerosi metodi di utilità per inserire rapidamente elementi in dizionari o liste JSON.

```java
public static void addTranslation(Namespace namespace, Locale locale,
String key, String value) {
      String formattedLocale = LocaleUtils.formatLocale(locale);
      JsonObject content = getOrCreateJsonFile(namespace,
              Lang.class,
              formattedLocale,
              () -> Lang.f.ofName(formattedLocale, "{}")
      ).getContent();
      content.addProperty(key, value);
  }
```

Codice 32: Applicazione del metodo `getOrCreateJsonFile()`

In questo esempio viene aggiunta una nuova traduzione per un determinato `Locale` [53] (lingua). La traduzione è rappresentata da una coppia chiave-valore, in cui la chiave identifica in modo univoco la componente testuale, e il valore ne specifica la traduzione per il `Locale` indicato. Il metodo ottiene il contenuto JSON del file lang corrispondente al `Locale` richiesto. Successivamente vi aggiunge la coppia chiave-valore. Nel caso in cui il file non esista ancora (ad esempio, alla prima esecuzione per quel `Locale`), esso viene creato tramite la factory, garantendo comunque l'esistenza del file di traduzione prima dell'inserimento dei dati.

Un'altra applicazione simile sono le funzioni `setOnTick()` e `setOnLoad()`, che permettono di aggiungere o un'intera `Function` o una stringa contenenti comandi alla lista di funzioni da eseguire ogni *tick* o ad ogni caricamento dei file.

È stato precedentemente menzionato che nel `Builder` di `Project`, in base alla versione specificata, si ottiene il *pack format* di *datapack* e *resourcepack*. Questi valori sono memorizzati in un `Record` [54] chiamato `VersionInfo`.

4.4.2. Ottenimento Versioni

Quando il `Builder` chiama `VersionUtils.getVersionInfo(String versionKey)`, dove `versionKey` rappresenta il nome della versione (ad esempio `25w05a`), esegue i seguenti passi:

1. controlla che sia presente nel *path* del progetto `resources/_generated` un file JSON contenente tutte le versioni e i format associati (`versions.json`);
2. controlla che sia passato più di un giorno dall'ultima volta che è stato scritto `versions.json`;
3. Se il file non è presente oppure è passato più di un giorno dall'ultima volta che è stata eseguita la generazione del file, e dunque c'è la possibilità che sia stata pubblicata una nuova versione o *snapshot*, si ricrea il file.
4. carica il file come `JsonObject`
5. se `versionKey=="latest"`, vuol dire che sta cercando la versione più recente,
   1. crea un `Iterator`[7] di `JsonObject` per ottenere il valore del primo elemento;
   2. converte l'elemento in un `Record` `VersionInfo`.

6. se invece `versionKey` è una chiave valida, viene restituito l'oggetto `VersionInfo` corrispondente alla chiave richiesta.

Ma come viene creato `versions.json`? Ogni volta che è necessario creare un nuovo file, viene fatta una chiamata HTTP [55] ad un'API [56] che restituisce un oggetto JSON contenente i dati di tutte le versioni.
Queste vengono poi mappate al nome della versione corrispondente e ordinate dalla più nuova alla più vecchia. La mappa cosi creata è avvolta in un `Optional`. Se quest'ultimo è vuoto verrà sollevato un errore, altrimenti si scriverà la mappa sul file `versions.json`.

4.4.3. Esportazione in File Compressi

datapack e *resourcepack* vengono letti ed eseguiti dal compilatore di *Minecraft* anche se compressi in archivi `.zip`. Questo formato è particolarmente adatto alla distribuzione, poiché permette di offrire agli utenti due pacchetti leggeri e separati da scaricare.
La classe `Project` dispone di un metodo `buildZip()`, che, dopo aver ottenuto le cartelle *datapack* e *resourcepack* tramite il metodo `build()`, provvede a comprimerle generando i rispettivi archivi `.zip`. Al termine dell'operazione, le cartelle originali vengono eliminate.

Il metodo `zipDirectory()` si occupa di comprimere il contenuto di una cartella in un archivio `.zip`. Esplora tutte le sottocartelle e file presenti nel percorso specificato, aggiungendo ciascun file all'archivio di destinazione.
Per farlo, utilizza il metodo `Files.walk(folder)`, che genera uno `stream` di tutti i percorsi contenuti nella cartella, escludendo le cartelle. Per ogni file trovato, viene calcolato il percorso relativo rispetto alla cartella base (`basePath`), in modo che all'interno dell'archivio venga mantenuta la stessa struttura del progetto originale.
Successivamente, il metodo apre uno `stream` di lettura sul file e crea una nuova *entry* ZIP, ovvero un elemento che rappresenta un singolo file all'interno dell'archivio. L'oggetto `ZipArchiveOutputStream` [57] della libreria `commons-compress` [58] si occupa di aprire l'*entry* per consentire la scrittura dei dati relativi al file. Il contenuto viene quindi copiato nell'archivio tramite la classe `IOUtils` [59] di *Apache Commons*, dopodiché l'*entry* viene chiusa per indicare che la scrittura del file è stata completata.

Il metodo `buildZip()` è stato pensato per essere usato in concomitanza con un *workflow* [60] di GitHub che, qualora il progetto abbia una *repository* associata, costruisce le cartelle compresse di *datapack* e *resourcepack* ogni volta che viene creata una nuova *release* [61]. Questi archivi, onde evitare confusione tra le versioni, vengono automaticamente nominati con la versione specificata nel file `pom.xml` [62] del progetto e saranno scaricabili dalla pagina GitHub che contiene gli artefatti associati alla *release*.

[7] Dato che un `Set` non è ordinato, non dispone di un metodo `getFirst()`, e dunque si ricorre all'`Iterator`.

4.5. Uso working example

In questo capitolo verrà implementato un progetto che utilizza la libreria per modificare un *item* di *Minecraft*. L'obiettivo è fare in modo che, al click con il tasto destro del mouse, l'oggetto consumi uno tra tre diversi tipi di munizioni (anch'esse nuovi item), generando un'onda sinusoidale la cui lunghezza varia in base al tipo di munizione utilizzata.

Viene innanzitutto creato il progetto:

```java
Project myProject = new Project.Builder()
    .projectName("esempio")
    .version("1.21.10")
    .worldName("OOPack test world")
    .icon("icona")
    .description("esempio tesi")
    .addBuildPath("C:\\Users\\Ale\\AppData\\Roaming\\.minecraft")
    .build();
```

In seguito si dichiara il *namespace* che verrà utilizzato:

```java
var namespace = Namespace.of("esempio");
```

Viene creato il modulo `Munizioni`, che si occuperà di definire il codice e le risorse degli oggetti che saranno consumati. L'*item* munizione non ha comportamenti propri, tuttavia dispone di una ricetta per poter essere creato a partire da altri *item*. Dunque, ho scritto un metodo `make()` che in base ai parametri passati crea le 3 munizioni diverse.

```java
@Override
protected void content() {
  make("blue_ammo", "Munizione Blu", "Blue Ammo", "diamond",20);
  make("green_ammo", "Munizione Verde", "Green Ammo", "emerald",25);
  make("purple_ammo", "Munizione Viola", "Purple Ammo",
  "amethyst_shard",30);
}
```

I parametri passati al metodo sono, nell'ordine: l'ID interno dell'*item*, la sua traduzione in italiano, la sua traduzione in inglese, l'ID di un altro *item* necessario per la sua creazione, e la distanza in blocchi del raggio generato dall'onda.

Il metodo `make()`, oltre ad aggiungere le traduzioni tramite i metodi di utilità:

```java
Util.addTranslation("item.esempio.%s".formatted(id), en);
Util.addTranslation(Locale.ITALY, "item.esempio.%s".formatted(id), it);
```

Crea i file relativi all'aspetto dell'*item*.

```java
Item.f.ofName(id,"""
        {
          "model": {
            "type": "minecraft:model",
            "model": "%s"
          }
        }
        """, Model.f.ofName("item/","""
        {
          "parent": "item/generated",
          "textures": {
            "layer0": "%s"
          }
        }
        """, Texture.of("item/"+id)
    )
);
```

Item Model Definition

Modello 3D

Texture

La funzione `makeData()` invece, si occupa di creare la *recipe*, ovvero il file JSON che indica come ottenere la munizione e le sue proprietà, tra cui la distanza dell'onda. Oltre alla *recipe*, è creato un *advancement* che si è soliti usare per rilevare quando un giocatore possiede uno degli ingredienti richiesti per la creazione dell'oggetto, e dunque comunica che la ricetta è disponibile tramite un messaggio sullo schermo.

Il modulo `MostraRaggio` si occupa di modificare un `carrot_on_a_stick`[8], per renderlo in grado di consumare le munizioni sopra create e mostrare l'onda.

Viene innanzitutto invocata una funzione che genera una *lookup table* contenente i valori necessari alla costruzione dell'onda. Questa memorizza i risultati della funzione seno per gli angoli da 0° a 360° moltiplicati per 10, in modo da rendere l'onda più marcata.

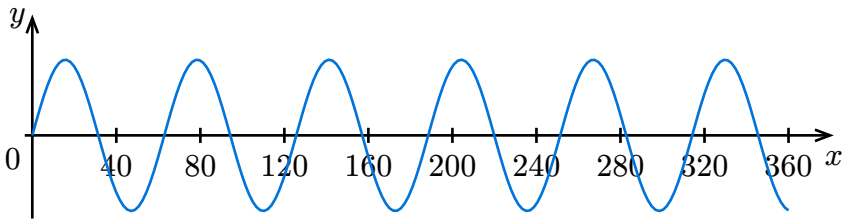

Figura 4: Rappresentazione dei valori memorizzati nella *lookup table*.

[8] `carrot_on_a_stick` è l'unico *item* che possiede una *scoreboard* in grado di rilevare quando è cliccato con il tasto destro.

```java
private void makeSinLookup() {
    StringBuilder sin = new StringBuilder("data modify storage
    esempio:storage sin set value [");
    for (int i = 0; i <= 360; i++) {
        sin.append("{value:").append(Math.sin(Math.toRadians(i *
        10))).append("},");
    }
    sin.append("]");
    Util.setOnLoad(Function.f.of(sin.toString()));
}
```

Successivamente si creano gli *scoreboard* utili al funzionamento del progetto. `click` aumenterà di 1 ogni volta che il giocatore clicca il tasto destro del mouse, mentre `var` è usato per le operazioni matematiche.

```java
Util.setOnLoad(Function.f.of("""
  scoreboard objectives add $ns$.click
  minecraft.used:minecraft.warped_fungus_on_a_stick
  scoreboard objectives add $ns$.var dummy
  """));
```

Il funzionamento dell'*item* è implementato con una catena di funzioni annidate. Alla radice c'è una funzione che ogni *tick* esegue la funzione (Codice 41) che sarà passata come `varargs` della factory, che sostituirà `%s`.

```java
1   var tick = Function.f.of("""
2     execute as @a at @s run function %s""",
…                                       …
47  );
48  Util.setOnTick(tick);
```

Codice 40

Questa funzione invoca Codice 42 se il giocatore ha cliccato l'*item*, e in seguito azzera il valore dello *scoreboard* per evitare che nel prossimo *tick* venga eseguita nuovamente la funzione anche se l'*item* non è stato usato.

```java
Function.f.of("""
  execute if score @s $ns$.click matches 1.. run function %s
  scoreboard players reset @s $ns$.click
""",
```

Codice 41

I seguenti comandi si occupando di controllare se il giocatore possiede *item* identificati come `ammo`. In caso negativo viene bloccato il flusso di esecuzione, in caso positivo viene invocata una funzione (Codice 43) per ottenere la prima munizione che il giocatore possiede. Se è stata trovata una munizione, viene eseguito Codice 44.

```java
Function.f.of("""
  execute unless items entity @s container.* *[minecraft:custom_data~{$ns$:
  {ammo:1b}}] run return fail
  data remove storage $ns$:storage item
  function %s
  execute if data storage $ns$:storage item run function %s
  """,
```

Codice 42

Questo metodo genera una `Function` che controlla i 36 *slot* del giocatore, arrestando l'esecuzione al primo *item* contrassegnato come `ammo`.

```java
private String getSlot() {
  var slots = new StringBuilder();
    for (int i = 0; i <= 35; i++) {
      slots.append("""
        execute if items entity @s container.%1$s
        *[minecraft:custom_data~{$ns$:{ammo:1b}}] run return run data modify
        storage $ns$:storage item set from entity @s Inventory[{Slot:
        %1$sb}]""".formatted(i));
    }
    return slots.toString();
}
```

Codice 43

Se l'*item* è stato trovato, vengono eseguiti i seguenti comandi:

1. Codice 44-2: salva la distanza associata alla munizione in una *scoreboard*;
2. Codice 44-3: viene riprodotto un suono. Tramite il metodo di utilità `addSound()` questo è aggiunto al dizionario di `sounds.json` e `Sound.of()` si occupa di prelevare il file `.ogg` al *path* indicato;
3. Codice 44-4 chiama una funzione *macro* che elimina la munizione trovata dallo *slot* corrispondente;
4. Codice 44-4 sposta l'esecuzione della funzione all'altezza degli occhi del giocatore, e invoca Codice 45.

```java
Function.f.of("""
  execute store result score $distance $ns$.var run data get storage
  $ns$:storage item.components."minecraft:custom_data".$ns$.distance 10
  playsound %s player @a[distance=..16]
  function %s with storage $ns$:storage
  item.components."minecraft:custom_data".$ns$
  execute anchored eyes positioned ^ ^ ^ run function %s
""", Util.addSound(
  "item.%s".formatted(id),
  "Beam Sparkles",
  Sound.of("item/%s".formatted(id)
)
```

Codice 44

La seguente funzione rappresenta il nucleo della logica ricorsiva per creare l'onda. Si rimuove 1 dallo *score* `distance`, e si memorizza l'esito di questa operazione in uno *storage*. Se ancora non si è raggiunta la distanza massima, ovvero `$ns$.var matches 1..` si sposta l'esecuzione 0.1 blocchi più avanti e si ripete la funzione.
Codice 45-4 invoca la funzione Codice 46, passando l'indice dell'iterazione corrente come parametro.

```java
Function.f.of("""
  scoreboard players remove $distance $ns$.var 1
  execute store result storage $ns$:storage distance.amount int 1 run
  scoreboard players get $distance $ns$.var
  function %s with storage $ns$:storage distance
  execute if score $distance $ns$.var matches 1.. positioned ^ ^ ^0.1 run
  function $ns$:$name$
""")
```

Codice 45

Questo comando *macro* invoca un'altra funzione *macro*, passandole il valore corrispondente a $\sin(\text{amount} \times 10)$.

```java
Function.f.of("""
  $function %s with storage esempio:storage sin[$(amount)]
""")
```

Codice 46

Questo valore è usato per determinare lo spostamento verticale della *particle*, dando quindi l'impressione che si stia disegnando una funzione sinusoidale.

```java
Function.f.of("""
  $particle end_rod ^ ^$(value) ^
""")
```

Codice 47

Successivamente i due moduli vengono registrati:

```java
Module.register(
  MostraRaggio.class,
  Munizioni.class
);
```

Uscendo dal *namespace* corrente, esso viene aggiunto indirettamente al progetto. Quest'ultimo viene costruito e generato in formato `.zip`:

```java
namespace.exit();
myProject.buildZip();
```

Sarà dunque possibile creare una *repository* e pubblicare una *release*. In seguito una *GitHub action* eseguirà il progetto per generare le due cartelle compresse e rinominarle. In questo caso si chiameranno `datapack-esempio-1.0.0.zip` e `resourcepack-esempio-1.0.0.zip`.

Conclusione

! TODO !

iniziare il capitolo con un riassunto del problema affrontato e della soluzione proposta.

Per misurare concretamente l'efficienza della libreria ho scritto una classe `Metrics` che si occupa di registrare il numero di righe e di file generati. Dopo aver eseguito il progetto associato al *working example*, si nota che il numero di file prodotti è 31, con un totale di 307 righe di codice.

Il codice sorgente dispone invece dei seguenti file Java[9]:

Classe	Righe di codice
`Main.java`	30
`MostraRaggio.java`	90
`Munizione.java`	100

Per un totale di 220 righe di codice in 3 file. Confrontando i due valori, si nota che il numero di file generati è pari a oltre dieci volte quello dei file sorgente. Le righe prodotte sono il 40% in più di quelle dei file sorgente.

[9] I valori riportati sono arrotondati al multiplo di dieci inferiore, al fine di escludere eventuali righe vuote o commenti.

Prendendo come riferimento un esempio più articolato, tratto da un progetto personale che implementa nuove piante, sono presenti i seguenti file:

Classe	Righe di codice
`Main.java`	170
`Seeds.java`	140
`Misc.java`	20
`Interaction.java`	100
`Heal.java`	90
`BloomingBulb.java`	290
`Bloomguard.java`	90
`EtchedVase.java`	300
`Blocks.java`	160

Eseguendo il programma, a partire da 9 file contenenti complessivamente 1360 righe di codice, vengono generati 137 file per un totale di 2451 righe.

Il seguente grafico mette in relazione il numero di righe e file prodotti per il *working example* (P_1) e il progetto appena citato (P_2).

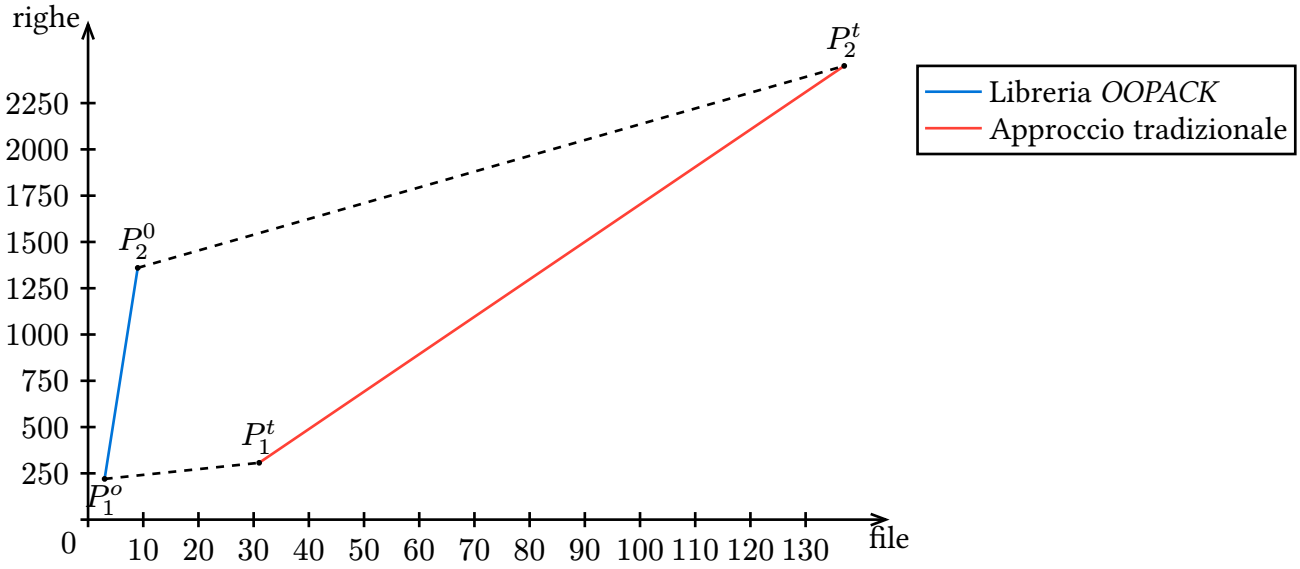

Figura 5: Numero di righe e file richiesti a confronto.

Si può osservare come la linea blu relativa alla libreria presenti una pendenza maggiore, evidenziando come il singolo file contenga molte più righe di codice.

Il vantaggio di utilizzare la libreria risulta particolarmente evidente nei progetti di ampia scala (P_2): una volta superata la fase iniziale in cui è necessario implementare metodi specifici per il progetto in questione, diventa immediato sfruttare la libreria per automatizzare la creazione di file simili.

Se si considera la distanza come il vantaggio tratto dall'utilizzo della libreria, è evidente che automatizzare lo sviluppo sia vantaggioso per i progetti di scala maggiore. Per un progetto piccolo come P_1, $d_1 = \sqrt{(3+31)^2 + (220+307)^2} = 528$.
Per P_2 invece, $d_2 = \sqrt{(9+37)^2 + (1360+2451)^2} = 3818$.

Se si misura la densità di codice per file come il rapporto tra righe totali e file totali, si vedrà che $p(P_1) = 73,7$ e $p(P_2) = 151,1$. Quindi, un raddoppio della densità del codice implica che il beneficio dell'automazione aumenta di oltre 7 volte. Dunque si può affermare che l'efficienza della libreria cresce in modo non lineare rispetto alla dimensione del progetto.

Devo tuttavia ammettere che, dopo aver iniziato a utilizzare questa libreria, è stato necessario un notevole sforzo mentale per lavorare con due linguaggi diversi contemporaneamente e sfruttarne appieno le potenzialità.

Non nego anche che ci sia la possibilità di aggiungere altri metodi di utilità, magari più specifici ma che potrebbero comunque ridurre la mole di lavoro a carico dello sviluppatore. Ad esempio un metodo che prende in input uno o più interi e crea la funzione contenente i comandi *scoreboard* che si occupano di inizializzare le costanti.

Oltre alle conoscenze tecniche acquisite, ritengo che lo sviluppo di questo progetto in un arco di tempo prolungato mi abbia permesso di riconsiderare alcune mie scelte implementative, andando oltre il semplice obiettivo di produrre software funzionante. Ho avuto l'opportunità di migliorare parti di codice che nonostante funzionassero correttamente, non rappresentavano la soluzione più efficiente o l'approccio più comodo per l'utente finale.
Questo processo di revisione mi ha aiutato a maturare un metodo di lavoro più critico alla qualità complessiva del software, ponendo particolare attenzione alla manutenibilità del codice e all'esperienza d'uso.

Bibliografia

[1] «Minecraft». [Online]. Disponibile su: https://minecraft.wiki/w/Minecraft

[2] «Mojang Studios». [Online]. Disponibile su: https://minecraft.wiki/w/Mojang_Studios

[3] «Sandbox Game». [Online]. Disponibile su: https://en.wikipedia.org/wiki/Sandbox_game

[4] «Minecraft Command». [Online]. Disponibile su: https://minecraft.wiki/w/Commands

[5] «Domain Specific Language». [Online]. Disponibile su: https://apice.unibo.it/xwiki/bin/view/TheFridge/DomainSpecificLanguage

[6] Ken Arnold, James Gosling, e David Holmes, *THE Java™ Programming Language*, Fourth Edition. [Online]. Disponibile su: https://www.acs.ase.ro/Media/Default/documents/java/ClaudiuVinte/books/ArnoldGoslingHolmes06.pdf

[7] «Minecraft Function». [Online]. Disponibile su: https://minecraft.wiki/w/Function_(Java_Edition)

[8] «JSON». [Online]. Disponibile su: https://www.json.org/json-en.html

[9] «Java Resource». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/technotes/guides/lang/resources.html

[10] «Resourcepack». [Online]. Disponibile su: https://minecraft.wiki/w/Resource_pack

[11] «Texture». [Online]. Disponibile su: https://invogames.com/blog/3-d-texturing-in-video-games-a-complete-guide/#what-is-3d-texturing-in-video-games

[12] «Portable Network Graphics». [Online]. Disponibile su: https://www.libpng.org/pub/png/book/chapter01.html

[13] «OGG Vorbis». [Online]. Disponibile su: https://it.wikipedia.org/wiki/Ogg

[14] «Game Assets». [Online]. Disponibile su: https://www.autodesk.com/uk/solutions/game-assets

[15] «Snapshot». [Online]. Disponibile su: https://minecraft.wiki/w/Snapshot

[16] «Namespace/Resource Location». [Online]. Disponibile su: https://minecraft.wiki/w/Resource_location

[17] «World Seed». [Online]. Disponibile su: https://minecraft.wiki/w/World_seed

[18] «Chunk». [Online]. Disponibile su: https://minecraft.wiki/w/Chunk

[19] «NBT Format». [Online]. Disponibile su: https://minecraft.wiki/w/NBT_format

[20] «Tick/Game Loop». [Online]. Disponibile su: https://minecraft.wiki/w/Tick

[21] «Particle». [Online]. Disponibile su: https://minecraft.wiki/w/Particles_(Java_Edition)

[22] «Turing completezza». [Online]. Disponibile su: https://www.cyfrin.io/glossary/turing-complete

[23] «Lookup Table». [Online]. Disponibile su: https://en.wikipedia.org/wiki/Lookup_table

[24] Guido van Rossum, *An Introduction to Python*, Release2.2.2 ed. [Online]. Disponibile su: https://scispace.com/pdf/an-introduction-to-python-1uphd66ueo.pdf

[25] «Symbolic Link e Junction». [Online]. Disponibile su: https://www.komprise.com/glossary_terms/symbolic-link/

[26] «Repository». [Online]. Disponibile su: https://docs.github.com/en/enterprise-cloud@latest/repositories/creating-and-managing-repositories/about-repositories

[27] «Git». [Online]. Disponibile su: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

[28] «GitHub». [Online]. Disponibile su: https://docs.github.com/en/get-started/start-your-journey/about-github-and-git#about-github

[29] «Minecraft Item». [Online]. Disponibile su: https://minecraft.wiki/w/Item

[30] «Inline Code». [Online]. Disponibile su: https://www.lenovo.com/us/en/glossary/inline/

[31] «Beet». [Online]. Disponibile su: https://github.com/mcbeet/beet

[32] «Superset». [Online]. Disponibile su: https://www.epicweb.dev/what-is-a-superset-in-programming

[33] «ANTLR». [Online]. Disponibile su: https://www.antlr.org/about.html

[34] «F-Strings». [Online]. Disponibile su: https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals

[35] «Java Package». [Online]. Disponibile su: https://www.w3schools.com/java/java_packages.asp

[36] «Path». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html

[37] «Predicate». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html

[38] «Optional». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

[39] «Set». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/util/Set.html

[40] «Stack». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html

[41] «Varargs». [Online]. Disponibile su: https://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html

[42] «Class». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html

[43] «StringBuilder». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

[44] «JsonObject». [Online]. Disponibile su: https://www.javadoc.io/doc/com.google.code.gson/gson/2.8.5/com/google/gson/JsonObject.html

[45] «GSON». [Online]. Disponibile su: https://github.com/google/gson/blob/main/README.md

[46] «JavaPoet». [Online]. Disponibile su: https://square.github.io/javapoet/javadoc/javapoet/

[47] «BufferedImage». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/awt/image/BufferedImage.html

[48] «Map». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

[49] «Supplier». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html

[50] «Stream». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

[51] «Lazy Loading». [Online]. Disponibile su: https://java-design-patterns.com/patterns/lazy-loading/#detailed-explanation-of-lazy-loading-pattern-with-real-world-examples

[52] «Lazy Loading Example». [Online]. Disponibile su: https://stackoverflow.com/questions/28818506/optional-orelse-optional-in-java

[53] «Locale». [Online]. Disponibile su: https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html

[54] «Record». [Online]. Disponibile su: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Record.html

[55] «HTTP». [Online]. Disponibile su: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Overview

[56] «API». [Online]. Disponibile su: https://aws.amazon.com/what-is/api/

[57] «ZipArchiveOutputStream». [Online]. Disponibile su: https://commons.apache.org/proper/commons-compress/apidocs/org/apache/commons/compress/archivers/zip/ZipArchiveOutputStream.html

[58] «Commons Compress». [Online]. Disponibile su: https://commons.apache.org/proper/commons-compress/

[59] «IO Utils». [Online]. Disponibile su: https://commons.apache.org/proper/commons-io/apidocs/org/apache/commons/io/IOUtils.html

[60] «GitHub Workflows». [Online]. Disponibile su: https://docs.github.com/en/actions/how-tos/write-workflows

[61] «Release». [Online]. Disponibile su: https://docs.github.com/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository

[62] «pom.xml». [Online]. Disponibile su: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html#What_is_a_POM.3F