ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INFORMATICA
-
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 INFORMATICA
-
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

Il *Domain Specific Language* (*DSL*) del videogioco svedese *Minecraft*, *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 non dispone di costrutti quali 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 che, a partire da un'analisi approfondita delle carenze e difetti di *mcfunction*, giunge 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 sfrutta il sistema di tipi di Java per fornire validazione statica, 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 in soluzioni alternative.

Un *working example* conferma l'approccio scelto: nel *pack* di esempio il codice scritto è ridotto del 40%, consolidando 31 file in 3 file sorgente, con miglioramenti significativi in termini di densità del codice e manutenibilità del progetto.

Desidero innanzitutto ringraziare il prof. Padovani dell'*Alma Mater Studiorum* Università di Bologna per la disponibilità e il prezioso supporto durante questo percorso, e per avermi dato l'opportunità di approfondire e lavorare con tecnologie di mio particolare interesse.

Ringrazio la mia famiglia per avermi sempre incoraggiato e sostenuto durante tutto il mio percorso formativo. Un ringraziamento speciale va a mia zia Lalli, che mi ha accolto in casa sua per tre anni, offrendomi un ambiente sereno in cui potermi dedicare agli studi.

Infine, un grazie di cuore ai miei cari amici Alessio, Daniele, Giovanni, Jacopo e Luca per le tante ore di studio trascorse insieme e per aver reso la mia vita universitaria più leggera.

Indice dei contenuti

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 bizzarre strutture a cubi in un mondo virtuale, si è rivelato presto essere l'ambiente dove per anni ho scritto ed eseguito i miei primi frammenti di codice utilizzando il suo sistema di comandi.
Motivato dalla mia acquisita abilità nel saper programmare con questo linguaggio di scripting non convenzionale, ho intrapreso con entusiasmo un percorso di studi in informatica.

Creato nel 2009 dallo svedese Markus Persson e sviluppato nel 2011 dall'azienda Mojang Studios [2], *Minecraft* è un famoso videogioco tridimensionale appartenente al genere *sandbox* [3], cioè caratterizzato dall'assenza di una trama predefinita, dove è il giocatore stesso a costruire liberamente la propria esperienza e gli obiettivi da perseguire.
Il gioco presenta un mondo composto da cubi formati da *voxel* (controparte tridimensionale del pixel) generati proceduralmente, dove i giocatori possono raccogliere risorse, costruire strutture, creare oggetti e affrontare creature ostili.

Minecraft è diventato il videogioco più venduto al mondo, perché non è semplicemente un prodotto di intrattenimento, ma un ambiente flessibile, accessibile, continuamente ampliato e sostenuto da una community globale che lo ha trasformato in un fenomeno culturale trasversale.

Figura 1: Un mondo di *Minecraft*.

Fin dalle sue origini, i creatori di *Minecraft* hanno messo a disposizione dei giocatori un insieme di comandi [4] che consentiva di aggirare gli ostacoli incontrati nella propria esperienza di gioco.
Con il tempo, tale sistema si è evoluto in un articolato linguaggio di configurazione e scripting basato su file testuali, costituendo di fatto un *Domain Specific Language* [5] (*DSL*) mediante il 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]. I DSL sono sviluppati 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 olandese specializzata nella creazione di ambienti di sviluppo integrati (*Integrated Development Environments*, IDE), descrive perfettamente chi sono gli utilizzatori del *Domain Specific Language* di *Minecraft*.

Minecraft è sviluppato in Java [6], ma questo DSL, chiamato *mcfunction* [7], adotta un paradigma completamente diverso. Essa non consente di introdurre nuovi comportamenti

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

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 sempre più complesse e originali. Tuttavia, il sistema presenta ancora diverse limitazioni, poiché una parte sostanziale della logica continua a essere implementata attraverso i file *mcfunction*, meno versatili rispetto a Java.

Il tirocinio accademico ha avuto come obiettivo la progettazione e realizzazione di un framework che semplifica lo sviluppo e la distribuzione di gruppi di file *mcfunction* e JSON tramite un ambiente di sviluppo unificato. Tale framework 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 funzionalità, viene eseguito il programma per ottenere una cartella «pacchetto» (*pack*) pronta per essere utilizzata. In questo modo lo sviluppo del pacchetto risulta più coerente e accessibile, permettendo di integrare *feature* di Java in questo DSL per facilitare la scrittura e la gestione dei file.

Nel capitolo successivo viene presentata la struttura generale del sistema di *pack*, descrivendone gli elementi costitutivi e il loro funzionamento. Segue un'analisi sistematica delle principali problematiche e limitazioni tecniche dell'infrastruttura, corredata da una rassegna critica delle più recenti soluzioni proposte. 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 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 da utilizzare. 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 la possibilità di sovrascrivere le *texture* e altri *asset* [14] del videogioco per renderle più affini ai propri gusti. Gli sviluppatori di *datapack* hanno poi iniziato ad utilizzare *resourcepack* per definire le risorse che il loro progetto avrebbe impiegato. Le *resourcepack* hanno

portata globale e vengono applicate a tutti i *save file*, ovvero su ogni mondo creato. Le cartelle *datapack*, invece, devono essere collocate nella directory `datapack` dei mondi nei quali si desidera utilizzarle.
Pertanto, partendo dalla cartella radice di *Minecraft* (`.minecraft/`), le *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 pubblicata dallo sviluppatore.

2.2. Struttura e Componenti di Datapack e Resourcepack

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

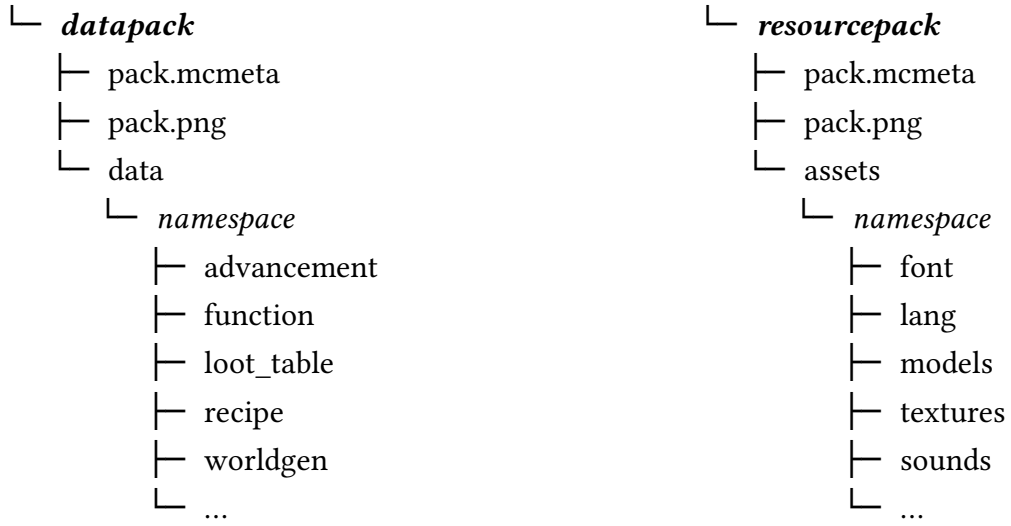

Figura 2: *datapack* e *resourcepack* a confronto.

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 del videogioco.

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 un *namespace* o coincide con il nome stesso del progetto che si sta sviluppando, o è una sua abbreviazione. *Datapack* e *resourcepack* adotteranno lo stesso *namespace*.
Tuttavia, si vedrà che operare in *namespace* distinti non è 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 alterarne il comportamento previsto. È interessante notare come anche gli sviluppatori di *Minecraft* stessi facciano uso dei *datapack* per definire e organizzare molti comportamenti del gioco, come ad esempio la dichiarazione delle risorse ottenibili dai blocchi scavati (*loot table*), o gli ingredienti necessari per creare un certo oggetto (*recipe*). 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 il quale il gioco stesso struttura e gestisce alcune delle sue funzionalità principali.
Occorre specificare che i comandi e i file `.mcfunction` non sono utilizzati in alcun modo dagli sviluppatori di *Minecraft* per implementare funzionalità del videogioco, dato che 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 file 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 determinati comandi.

[2] Sviluppatore di *Minecraft* inglese, membro del team che sviluppa *feature* inerenti a *datapack*.

Per identificare univocamente le risorse all'interno di *datapack* e *resourcepack* si utilizzano le *resource location*. La loro struttura è composta da due parti separate dal carattere `:`, il *namespace* seguito dal percorso della risorsa. Rispetto a un *path* completo, la *resource location* omette la cartella funzionale che categorizza il tipo di risorsa.
Ad esempio, per riferirsi alla ricetta situata nel percorso `foo/recipe/my_item.json`, si utilizza la *resource location* `foo:my_item`, dove `foo` è il namespace e `my_item` è l'identificatore della risorsa. La cartella `recipe`, che indica la tipologia della risorsa, non compare nella *resource location* poiché il compilatore determina automaticamente il tipo di risorsa in base al contesto d'uso. Se la *resource location* viene letta in un contesto che richiede una ricetta, il compilatore cercherà il file nella cartella `recipe`; se invece il contesto richiede una funzione, cercherà nella cartella `function`.

2.3. I Comandi

Prima di spiegare la funzione dei comandi, è necessario definire gli elementi basilari su cui essi agiscono.
Minecraft permette di creare ed esplorare mondi generati da un *seed* [17] casuale, ognuno diverso dagli altri. 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 nel 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 ai giocatori 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 richiedono 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.

Nonostante 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 associato ad 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 calcolare il risultato delle 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

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

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 utilizza 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 un prefisso, che convenzionalmente coincide con il *namespace*. Vengono dunque salvati nel file `command_storage_<namespace>.dat` come un dizionario NBT.

```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` permette l'esecuzione in catena di più comandi, cambiando valori quali l'entità esecutrice e la posizione. Questi e altri 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 identificare l'entità del contesto di esecuzione corrente.

Tramite `execute` è possibile specificare condizioni preliminari e memorizzare l'esito di un comando. 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 posizionata 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, esse non prevedono valori di input o di output, ma contengono uno o più comandi eseguiti secondo l'ordine in cui sono scritti nel file.

In base alla complessità del *branching* e alle operazioni eseguite dalle funzioni, il compilatore (o più precisamente, il motore di esecuzione dei comandi) alloca 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*.

Le funzioni possono essere invocate da altri file di un datapack in più modi:

- 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 in *Minecraft* vengono eseguite all'interno di un game loop, completando tutti i comandi che contengono, comprese eventuali chiamate ad altre funzioni. Quando un comando `execute` modifica il contesto di esecuzione (ad esempio cambiando il giocatore o la posizione), questa modifica non influenza i comandi successivi nella funzione corrente, ma si applica 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*.

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` per indicare insuccesso 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.

[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.

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 degli *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 concisa.

Problemi Pratici e Limiti Tecnici

Il linguaggio *mcfunction* non è stato originariamente concepito come un linguaggio di programmazione Turing completo. Infatti, negli anni antecedenti all'introduzione dei *datapack*, il comando `scoreboard` era impiegato in maniera convenzionale per monitorare le statistiche dei giocatori, quali 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 a giungere, nel 2017, alla nascita dei *datapack*.

Ancora oggi l'ecosistema dei *datapack* è in costante evoluzione, con *snapshot* che introducono nuove funzionalità o aggiornano quelle esistenti. Tuttavia, questo ambiente presenta ancora diverse limitazioni di natura tecnica, riconducibili al fatto che non era stato originariamente concepito per supportare logiche di programmazione complesse o per essere impiegato in progetti di grandi dimensioni.

3.1. Limitazioni di Scoreboard

Come è stato precedentemente citato, `scoreboard` è usato per eseguire operazioni su interi. Tuttavia, questo comando presenta numerosi vincoli.

Dopo aver creato un *objective*, è necessario impostare le costanti da utilizzare per le eventuali operazioni di moltiplicazione e divisione. Inoltre, è ammessa una sola operazione per comando `scoreboard`.

Di seguito viene mostrato come l'espressione `int x = (y*2)/4-2` si calcola in *mcfunction*. Le variabili sono 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 come 0 e il risultato dell'espressione non sarebbe stato corretto.

Si noti come, nell'esempio precedente, le operazioni vengano eseguite sulla variabile y, il cui valore risultante viene successivamente 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 esso rimane 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 di voler calcolare il 5% di 40. In un linguaggio di programmazione di alto livello, entrambe le espressioni `40/100*5` e `40*5/100` restituiscono correttamente il valore 2. Scomponendo queste operazioni in comandi `scoreboard` si ottengono rispettivamente i seguenti comandi:

```mcfunction
1 scoreboard players operation set $val math 40
2 scoreboard players operation $val math /= #100 math
3 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 le *scoreboard* è possibile eseguire esclusivamente le quattro operazioni aritmetiche fondamentali, il calcolo di funzioni più complesse, quali logaritmi, esponenziali, radici quadrate o funzioni trigonometriche, risulta particolarmente difficile da implementare.

Occorre inoltre considerare che tali operazioni sono limitate al dominio dei numeri interi. È dunque richiesto implementare un algoritmo che approssimi queste funzioni, oppure utilizzare 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 richiede spesso di gestire un input moltiplicato per 10^n con output (nell'esempio, il file `.out` della funzione `math` della *scoreboard*) di tipo intero le cui ultime n cifre rappresentano la parte decimale del risultato[5]. Inoltre, questo approccio può facilmente provocare problemi di *integer overflow*.

In seguito all'introduzione delle *macro*, si è diffuso l'utilizzo di *lookup table*. Una *lookup table* consiste in un *array* memorizzato in uno *storage* che contiene tutti gli output di una funzione per un intervallo prefissato di input.

Si ipotizzi sia richiesta la radice quadrata con precisione decimale di tutti gli interi tra 0 e 100. Si può creare uno *storage* che contenga 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 $n = 3$.

Poiché sono richiesti gli output per decine, se non centinaia, di valori in input, i comandi per la creazione delle *lookup table* sono generati mediante script Python [24] ed eseguiti dal *Minecraft* esclusivamente durante l'inizializzazione del *datapack* (tramite `load.json`). Dal momento che tali strutture sono soggette a sole operazioni di lettura e non di scrittura, non sussiste il rischio di modifiche durante la sessione di gioco.

3.3. Alto Rischio di Conflitti

In Codice 12 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 sovrascritture di dati.

Un'altra situazione che può provocare conflitti si verifica quando due *datapack* sovrascrivono la stessa risorsa nel *namespace* `minecraft`. Se entrambi modificano `minecraft/loot_table/blocks/stone.json`, che determina gli oggetti ottenibili da un blocco di pietra, il compilatore utilizzerà il file del *datapack* caricato per ultimo, ignorando le funzionalità dell'altro.

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*.

Nell'esempio seguente vengono presentati due frammenti di codice tratti da *datapack* sviluppati da autori diversi con il medesimo obiettivo di eseguire una funzione sull'entità chiamante (`@s`) al termine di un determinato intervallo di tempo. In entrambi i casi, le funzioni deputate all'aggiornamento del timer vengono eseguite a ogni *tick*, ossia 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 il medesimo *fakeplayer* all'interno della stessa *scoreboard*. Poiché `timer_a` incrementa il valore di `timer` mentre `timer_b` lo decrementa, al termine di ogni *tick* esso risulta invariato. Qualora entrambe modificassero `timer` nella stessa direzione, ad esempio incrementandolo, la durata effettiva del timer risulterebbe dimezzata. Questo costituisce 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 in base alle quali i selettori possono filtrare le entità, vi sono i *tag*, stringhe memorizzate in un array nei dati NBT di un'entità.
Dunque, 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 iterativamente vengono racchiusi tra parentesi graffe. In Python, invece, la stessa funzione è ottenuta tramite l'indentazione del codice.

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

[6] Come separatore si utilizza `.` anziché `:` in quanto quest'ultimo è un carattere ammesso nel nome di una *scoreboard*.

la medesima condizione su ciascuna riga. Quest'ultima soluzione comporta un maggiore *overhead*, in particolare quando il comando viene eseguito ripetutamente nel corso di più *tick*.

Di seguito viene illustrato un esempio di implementazione di un blocco `if-else` o `switch`, mediante l'utilizzo del comando `return` per interrompere il flusso di esecuzione nella funzione corrente.

conditional_example.mcfunction mcfunction

```
1  execute if entity @s[type=cow] run return run say I'm a cow
2  execute if entity @s[type=cat] run return run say I'm a cat
3  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 avanti non verranno eseguiti qualora il tipo dell'entità sia `cow`. Se la condizione alla riga 1 risulta falsa, l'esecuzione 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.

La funzione è sufficientemente intuitiva e simile a costrutti tipici dei linguaggi di programmazione di alto livello.

```
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.

Si ipotizzi ora di voler eseguire due o più comandi in base all'entità selezionata.

```
1  execute if entity @s[type=cow] run say I'm a cow                    mcfunction
2  execute if entity @s[type=cow] run return run say moo
3
4  execute if entity @s[type=cat] run say I'm a cat
5  execute if entity @s[type=cat] run return run say meow
6
7  say I'm neither a cow or a cat
```

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

Si noti come la condizione da soddisfare per l'esecuzione dei comandi sia ripetuta, in quanto non è possibile raggrupparli in un blocco di codice. Tale approccio comporta un notevole overhead, specialmente per operazioni di selezione dispendiose quali la deserializzazione di NBT. Dunque si creano funzioni che raggruppano i comandi da eseguire sotto la stessa condizione.

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 condizionale di funzioni in base a eventi naturalmente occorrenti nel gioco, sono numerosi i casi in cui ci si trova a creare più file che contengono un numero ridotto, purché significativo, di comandi.

L'unica strategia per implementare cicli è mostrata in Codice 7, attraverso la ricorsione. Di conseguenza, ogni volta che è necessario implementare un ciclo, è indispensabile creare almeno una funzione che si richiama. Se è invece richiesto 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*.

Un'entità giocatore dispone di 36 *slot* utilizzati per contenere oggetti. Si ipotizzi di voler individuare in quale *slot* dell'inventario del giocatore si trovi l'oggetto `diamond`. Una possibile soluzione consiste nell'utilizzare una funzione che iteri da 0 a 35, dove il parametro della *macro* indica lo *slot* da controllare. Tuttavia, questo approccio comporta un *overhead* maggiore rispetto alla verifica esplicita di ciascuno dei 36 *slot*.

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.

Il Codice 6 illustra come l'impiego delle *macro* imponga la definizione di una funzione dedicata: tale funzione deve essere in grado di accettare parametri esterni e di sostituirli nei comandi contrassegnati dal simbolo `$`. Si tratta verosimilmente dell'unico caso in cui la creazione di una nuova funzione risulta genuinamente giustificata e non causata da vincoli di *mcfunction*.

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

Ciò comporta un numero di file sproporzionato rispetto alle effettive righe di codice. Inoltre, si presentano ulteriori problematiche relative alla struttura delle cartelle e dei file nello sviluppo di *datapack* e *resourcepack*.

3.5. Organizzazione e Complessità della Struttura dei File

Le limitazioni precedentemente illustrate sono inerenti alla sintassi dei comandi e ai limiti delle funzioni; tuttavia, non sono da trascurare le complessità legate all'organizzazione e alla struttura 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 cartelle in modo separato può risultare oneroso, considerando l'elevato grado di interdipendenza tra le due. 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 contenente sia il *datapack* sia la *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 al *datapack* e alla *resourcepack* pubblicati, utilizzabile nei principali siti di hosting e condivisione di *pack*. Tali piattaforme, gestite da piccoli team di sviluppo, tendono ad affidarsi a servizi esterni per l'archiviazione dei file, come appunto GitHub.

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

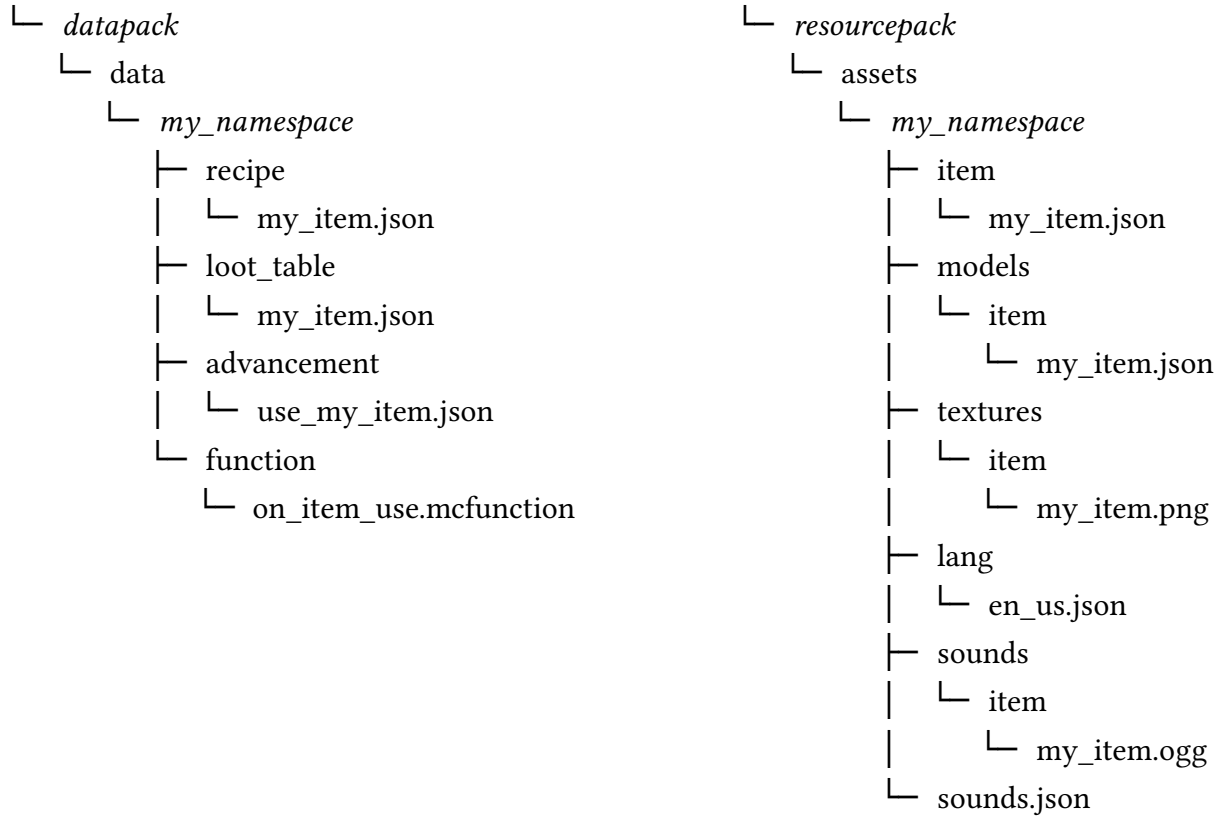

Figura 3: File necessari per implementare un semplice *item*.

Nella sezione *data*, che determina la logica e i contenuti del gioco, `loot_table` e `recipe` definiscono rispettivamente gli attributi dell'oggetto e la modalità con cui questo può essere creato. L'*advancement* `use_my_item` è usato per rilevare quando un giocatore usa l'oggetto, invocando la funzione `on_item_use` che in questo esempio riprodurrà un suono.

I suoni devono essere collocati nella directory *assets*. Affinché possano essere riprodotti, i file audio in formato `.ogg` devono essere registrati nel file `sounds.json`. Nella cartella *lang* sono 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 l'utilizzo di una o più risorse direttamente all'interno del 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 intuitivo.

In primo luogo, gli stessi sviluppatori di *Minecraft* dispongono di strumenti interni che automatizzano la creazione dei file JSON necessari al corretto funzionamento di determinate *feature*. Durante lo sviluppo, tali file vengono generati 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`, il quale registra i suoni e definisce quali file `.ogg` utilizzare. Questo contiene quasi 25.000 righe di oggetti JSON, ed è creato e aggiornato tramite software appositi ogni volta che viene inserita una *feature* che necessita di 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* con mezzi più pratici e 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 comprendere direttamente il codice sorgente di *beet*.

Altri precompilatori forniscono un'interfaccia più intuitiva e un utilizzo più immediato al costo della 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`.

Progettazione della Libreria

4.1. Approccio al Problema

Alla luce del contesto descritto e delle limitazioni degli strumenti esistenti, si è ricercata una soluzione che consentisse di ridurre la complessità e preservare la completezza delle funzionalità. Di seguito sono illustrate le principali decisioni progettuali e le ragioni che hanno portato alla scelta del linguaggio di sviluppo.

Inizialmente si è tentato di progettare un *superset* [32] di *mcfunction*, ovvero un linguaggio che estende quello originale introducendo nuove funzionalità, e mantenendone la compatibilità. Tale linguaggio avrebbe consentito di dichiarare e utilizzare elementi multipli (*mcfunction* e JSON) all'interno di un unico file, arricchendo inoltre la sintassi dei comandi con zucchero sintattico volto a velocizzare la scrittura delle sezioni 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`

Compilando il codice di questo linguaggio ideale, verrebbe non solo creata la funzione definita all'interno delle parentesi graffe, ma anche inserito il *namespace* prima di `var` e verrebbe creato il comando che assegna allo *score* costante `#4` il 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
```

Inizialmente si è scelto di utilizzare la versione Java della libreria ANTLR [33] per definire la grammatica del linguaggio. Tuttavia, è emerso che la realizzazione di una grammatica in grado di cogliere tutte le sfumature della sintassi di *mcfunction*, integrandovi al contempo le estensioni proposte, avrebbe richiesto un impegno di sviluppo incompatibile con i vincoli temporali di un progetto di tirocinio.

Si è quindi optato per lo sviluppo di una libreria che consentisse di definire la struttura di un *pack*, dalla radice del progetto fino ai singoli file, mediante oggetti, in modo da rappresentare l'intero insieme delle risorse come un albero n-ario. Tale struttura viene quindi attraversata in fase di esecuzione 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 coman-

di 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.

Si è dunque valutato quale linguaggio di programmazione, tra Python e Java, fosse più adatto per la realizzazione della 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.

A seguito di un'attenta analisi, si è optato per Java come linguaggio di sviluppo del progetto, in quanto esso consente di applicare *design pattern* volti a semplificare e rendere più robusta l'implementazione, pur a scapito di una minore immediatezza 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 utilizzati dall'utente finale.

`objects` Contiene le classi che rappresentano gli oggetti impiegati da *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 la struttura di un *pack* (`Datapack`, `Resourcepack`, `Namespace`, `Project`), a disposizione dell'utente finale.

Questa libreria si occupa di generare *pack*, e dunque non interviene direttamente sul codice sorgente Java di *Minecraft* per introdurre comportamenti dinamici. Tuttavia consentirà di facilitare la creazione di molteplici componenti statiche che, selezionate tramite comandi o altri elementi di un *datapack*, emulano un funzionamento dinamico. Un esempio emblematico è rappresentato dalle *lookup table*: i valori non vengono calcolati a runtime, bensì pre-generati a compile time per essere recuperati successivamente.

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()`, della classe di più alto livello, `Project`. Di conseguenza, ogni oggetto appartenente al progetto deve essere *buildable*, ovvero «costruibile», in modo da poter generare il file corrispondente in base al proprio nome e contenuto. 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 specifica la directory di destinazione nel file system locale nella quale verrà generato il file. Durante il processo di costruzione del progetto, questo percorso viene progressivamente esteso aggiungendo sotto-cartelle, 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*. Essa definisce il contratto `getContent()`, che specifica il contenuto associato all'oggetto. Sfruttando il polimorfismo, tale metodo può restituire il contenuto delle classi che rappresentano file, oppure un insieme di `FileSystemObject` per le classi che rappresentano cartelle o altri contenitori.

L'interfaccia `FileSystemObject` implementa il *design pattern* strutturale *composite*. Questa architettura permette di organizzare gli oggetti in strutture ad albero, garantendo una gestione uniforme sia per le singole istanze che per le loro aggregazioni.

Questa interfaccia definisce il metodo statico `find()`, il quale permette di trovare un `file` all'interno di un `FileSystemObject` che soddisfa una certa condizione, permettendo la comunicazione tra `FileSystemObject` in ogni punto del progetto.

```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 accetta come parametri un `FileSystemObject` (senza distinzione tra cartella o file), la classe del tipo ricercato (`clazz`) e un `Predicate` [37] che esprime la condizione da soddisfare.

Il metodo implementa un algoritmo di ricerca ricorsiva in profondità sulla struttura ad albero. Per ogni nodo visitato, verifica innanzitutto se esso è un'istanza del tipo ricercato; in tal caso, valuta il predicato fornito e, se soddisfatto, restituisce un `Optional` [38] contenente l'oggetto. Qualora il nodo corrente non corrisponda al tipo ricercato, il metodo ne recupera il contenuto: se questo è un `Set` [39], indicando che si tratta di una cartella o una sua sotto-classe, il metodo viene invocato ricorsivamente su ciascun elemento figlio, interrompendo la ricerca non appena viene trovata una corrispondenza. In assenza di risultati, viene restituito un `Optional` vuoto.

L'interfaccia `FileSystemObject` definisce inoltre il contratto `collectByType(Namespace data, Namespace assets)`, il quale viene sovrascritto dalle classi concrete per specificare l'appartenenza alla categoria *data* dei *datapack* o *assets* delle *resourcepack*.

Questo è un esempio di utilizzo del *design pattern* comportamentale *strategy*. Esso permette di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. In questo caso viene applicato per separare automaticamente le risorse sfruttano il polimorfismo.

4.2.2. AbstractFile e AbstractFolder

Tutti gli oggetti rappresentanti file nel progetto, il cui metodo `build()` scriverà in memoria, sono un'estensione della classe `AbstractFile`.
La classe astratta `AbstractFile<T>` è parametrizzata con un tipo generico `T`, relativo al contenuto del file, memorizzato nell'attributo `content`. La classe dispone dell'attributo `name`, nel quale è memorizzato il nome del file associato, privo di estensione. Possiede inoltre un riferimento al `parent`, ovvero alla sottocartella o cartella delle risorse in cui il file sarà collocato. L'oggetto dispone infine di un riferimento al *namespace* in cui è contenuto.
Il metodo `toString()` combina gli attributi `namespace` e `name` per generare la stringa corrispondente alla *resource location* dell'oggetto. Grazie a questa implementazione, è possibile inserire direttamente la variabile che rappresenta il file all'interno di altre stringhe, e la sua *resource location* viene individuata tramite *casting* implicito a stringa.

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

La classe `AbstractFile`, oltre ad implementare `FileSystemObject`, implementa le interfacce `PackFolder` ed `Extension`.
L'interfaccia `PackFolder` fornisce un unico contratto, `getFolderName()`, per definire il nome della cartella in cui l'oggetto sarà collocato. Ad esempio, la classe `Function` implementa tale metodo restituendo la stringa `"function"`, poiché tutte le funzioni devono risiedere nella cartella `function`.
Similmente, l'interfaccia `Extension`, mediante il contratto `getExtension()`, consente agli oggetti che estendono `AbstractFile` di specificare la propria estensione (`.json`, `.mcfunction`, `.png`).

L'altra classe astratta che implementa `FileSystemObject` è `AbstractFolder`, parametrizzata con il tipo generico `<T extends FileSystemObject>`. Tale classe mantiene un attributo `children` di tipo `Set<T>`, usato per memorizzare i riferimenti ai nodi figli garantendo l'unicità degli elementi. Il metodo `build()` implementa un attraversamento ricorsivo invocando `build()` su ciascun nodo contenuto in `children`.
In maniera analoga, il metodo `collectByType(...)` propaga ricorsivamente la classificazione degli oggetti attraverso l'albero, effettuando chiamate polimorfiche a `collectByType(...)` su ogni nodo figlio.

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 libreria, ma non è pensato per l'utilizzo dell'utente finale.

Nella prima iterazione 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 istanziati tramite l'operatore `new`.
Un sistema basato sulla creazione diretta degli oggetti presenta tuttavia 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, specialmente considerando l'obiettivo di offrire un'interfaccia più semplice e immediata per la creazione dei file.

Il sistema è stato quindi modificato per appoggiarsi su un oggetto `Context` che rappresenta il *parent*, ovvero la cartella a cui si vogliono aggiungere nodi. La classe `Context` contiene un attributo statico e privato di tipo `Stack<ContextItem>` [40], utilizzato per tracciare il livello di *nesting* delle cartelle. Il metodo `stack.peek()` restituisce il `ContextItem` in cima allo stack, corrispondente al contesto corrente.

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 fornisce inoltre due metodi `default` i quali permettono di inserire o eliminare l'oggetto dal `Context`.

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

Codice 25: Metodi dell'interfaccia `ContextItem`.

Invocando `enter()`, si inserisce l'oggetto che implementa `ContextItem` in cima allo `stack` del contesto, indicando che i prossimi `FileSystemObject` saranno inseriti in esso. 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. Factory

Il sistema deve garantire ad ogni oggetto che estende `FileSystemObject` di essere collocato nel `ContextItem` corretto. Per gestire automaticamente questo aspetto e al tempo stesso evitare la creazione diretta tramite `new`, si ricorre al *design pattern factory*.

Le *factory* costituiscono un *design pattern* creazionale finalizzato a separare la logica di inizializzazione degli oggetti dal codice che li utilizza. Anziché istanziare le classi direttamente, il client delega alla *factory* la creazione dell'oggetto desiderato. La *factory* si occupa di selezionare la classe concreta da istanziare e di determinarne lo stato iniziale. Nell'implementazione proposta, la *factory* gestisce inoltre l'inserimento dell'oggetto appena creato 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.

Tale approccio risulta particolarmente vantaggioso poiché permette di fornire all'utente molteplici funzioni dedicate alla istanziazione di oggetti.

```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 definire esplicitamente il nome del file oppure affidare alla libreria la generazione di un identificatore automatico. Come avviene nei compilatori convenzionali, i dettagli implementativi del codice generato e la nomenclatura dei file risultano irrilevanti per l'utente, il quale si limita a verificarne il corretto funzionamento senza necessità di ispezionare gli artefatti prodotti.

Qualora la stringa `name` passata come parametro contenga uno o più caratteri `/`, questi saranno interpretati come separatori di cartelle, creando una gerarchia di sottocartelle.
Il nome assegnato all'oggetto non influisce sul funzionamento della libreria, poiché quando questo viene utilizzato in un contesto testuale, la chiamata implicita al metodo `toString()` restituisce la sua *resource location*.

```java
Namespace namespace = Namespace.of("foo");
Function function = Function.f.ofName("bar/baz/my_function","say hello world!");
System.out.println(function);
```

Questo esempio stamperà `foo:bar/baz/my_function`, ovvero la *resource location* della funzione creata.

Gli oggetti passati come parametro *variable arguments* (*varargs* [41], `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 file di contenuto testuale.
Questa *nested class*, chiamata `Factory`, dispone di due parametri e ha il compito di istanziare le sottoclassi di `PlainFile`.

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

Codice 28: Intestazione della classe `Factory` per `PlainFile`

`F` è un tipo generico che estende `PlainFile<C>`, 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 determinato dalle sottoclassi che ereditano `PlainFile`. Ciò consente alla *factory* di operare in modo generico, generando file con contenuti eterogenei senza necessità di duplicare codice.

La *factory* mantiene un riferimento all'oggetto `Class` [42] parametrizzato con il tipo `F`, corrispondente alla classe degli oggetti da istanziare, utilizzato nel metodo `instantiate()`. Questa funzione restituisce l'oggetto da creare dati due parametri: il nome del file da creare e il suo contenuto di tipo `Object`, in quanto si sta ancora operando in un contesto generico.

Per istanziare l'oggetto, la funzione ottiene inizialmente un riferimento alla classe del contenuto (`StringBuilder.class` o `JsonObject.class`), necessario per individuare il costruttore della classe `F`. Successivamente, recupera il costruttore tramite *reflection*, verificando che la classe `F` disponga di un costruttore con i parametri `String name` e `C content`. Prima di procedere con l'istanziazione, rende accessibile il costruttore, operazione indispensabile per accedere a costruttori privati o protetti. In seguito, crea un'istanza della classe e la aggiunge al contesto corrente. Infine, restituisce l'oggetto creato.

Le classi `TextFile` e `JsonFile` estendono `AbstractFile`, utilizzando rispettivamente `StringBuilder` [43] e `JsonObject` [44] come tipo di `content`.

`TextFile` rappresenta un file di testo generico, il cui contenuto è gestito tramite un oggetto `StringBuilder` per consentire la concatenazione efficiente di stringhe. 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 astratta ereditata da tutti i file JSON 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 specifici per la creazione di file JSON.

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

Codice 29: 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"`.

Nonostante il contenuto richiesto dalle classi sopra descritte non sia di tipo `String`, esso viene comunque convertito in stringa prima della scrittura su file.

Prima della scrittura effettiva, ogni file testuale viene sottoposto a un leggero processo di *parsing*. Oltre alla già citata sostituzione dei valori segnaposto `%s`, dopo che `StringBuilder` e `JsonObject` sono stati convertiti in stringhe, il contenuto viene analizzato per individuare pattern specifici. La sottostringa `"$ns$"` viene sostituita con il nome effettivo del *namespace* attivo al momento della costruzione, mentre `"$name$"` viene sostituito con la *resource location* del file. 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.

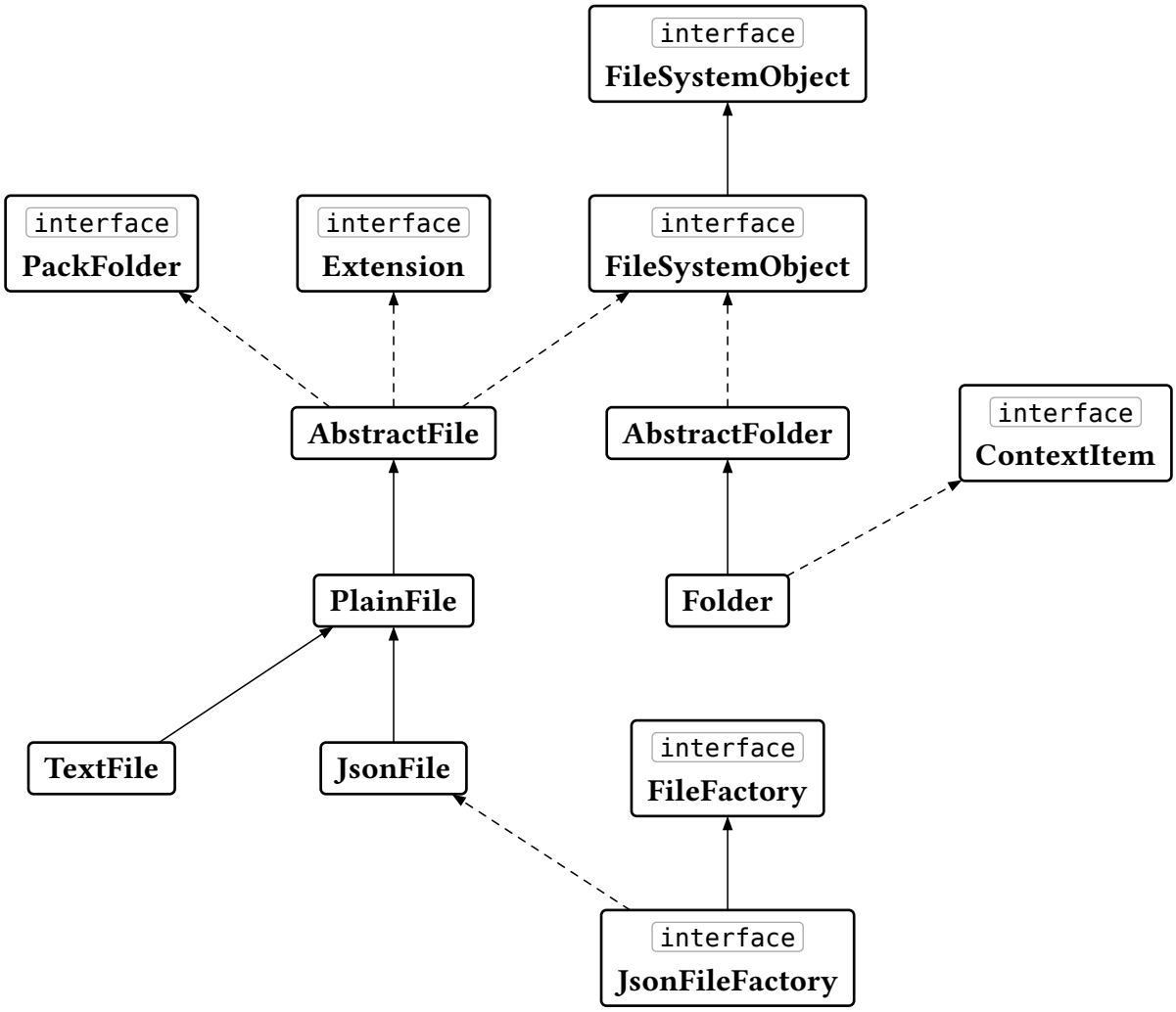

Figura 4: Diagramma del sistema progettato fino a questo punto.

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 altri DSL caratterizzati 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

Le classi astratte `DataJson` e `AssetsJson`, sottoclassi di `JsonFile`, eseguono l'*override* del metodo `collectByType()` di `FileSystemObject` per specificare se il file rappresentato appartiene rispettivamente alla categoria *datapack* o *resourcepack*.

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

Codice 30: metodo `collectByType()` di `DataJson`.

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

Unica eccezione è la classe `Function`. Questa estende direttamente `TextFile`, indicando la propria estensione (`.mcfunction`) con *override* del metodo `getExtension()`, e il tipo tramite *override* di `collectByType()` similmente a `DataJson`. Dal momento 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 un attributo statico e pubblico di tipo `JsonFileFactory<...>` chiamato `f`, parametrizzato per la classe specifica che istanzia. Con questo approccio si ha accesso rapido ai *factory methods* di ogni file, JSON e non. Queste classi sono 39 in totale, e ognuna corrisponde a uno specifico oggetto utile al funzionamento di un *datapack* o *resourcepack* (30 e 9 rispettivamente). Poiché ognuna di queste deve disporre di una *factory*, un costruttore, ed eseguire l'*override* del metodo `getFolderName()`, è stata impiegata una libreria per generare il loro codice Java.

Un possibile approccio alternativo avrebbe previsto l'implementazione di un metodo statico generico all'interno di `JsonFile.Factory`, strutturato per accettare come argomenti il tipo della classe da istanziare e la relativa directory di riferimento. Così facendo non sarebbe stato necessario creare una classe dedicata per ciascun tipo di file, ma sarebbe risultato sufficiente invocare direttamente la funzione `create()` per generare l'istanza desiderata.

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

Codice 31: Esempio di approccio alternativo.

Tuttavia è evidente che non risulta comodo per l'utente finale dover specificare tutti questi parametri ogni volta che necessita di utilizzare la *factory*.
Dunque è stata implementata una classe di utilità `CodeGen` che sfrutta la libreria *JavaPoet* [46] per creare le classi che rappresentano i file di un *pack* e i metodi al loro interno. Con queste classi, per creare un modello si può semplicemente scrivere `Model.f.of(json)`.

Classi che rappresentano file binari (immagini, suoni) 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`).

4.3.2. Module

È stata definita una sottoclasse astratta di `Folder`, denominata `Module`, con l'obiettivo di promuovere la modularità del codice attraverso una chiara separazione delle responsabilità e l'aggregazione di contenuti affini. Ad esempio, nel contesto dell'implementazione di una feature A, tutte le risorse e i dati ad essa correlati possono essere raggruppati all'interno dello specifico `Module` A.

La classe dispone di un *entry point*, ovvero una funzione astratta `content()` che verrà sovrascritta da tutte le classi che ereditano `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)`, il quale 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.

4.3.3. Namespace

Le classi concrete di file sono raggruppate all'interno di un `Namespace`. Analogamente alla classe `Folder`, quest'ultimo gestisce un `Set` di elementi figli e implementa le interfacce `Buildable` e `ContextItem`. L'implementazione di quest'ultima è necessaria poiché un `Project` può essere composto da molteplici *namespace*; è pertanto indispensabile tracciare

quello corrente destinato ad accogliere i `FileSystemObject` istanziati.
Poiché gli elementi figli di `Namespace` sono di natura diversa (*data* o *assets*), è necessario dividerli prima della loro scrittura su file. Questi devono essere indirizzati verso i rispettivi contesti: il *namespace* del *datapack* per la componente *data* e quello relativo alle *resourcepack* per gli *assets*.

La classe presenta una particolarità nel suo metodo `exit()`, usato per indicare 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.

4.3.4. Project

La classe `Project` rappresenta la radice dell'albero corrispondente all'intero *pack*, 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 il costruttore di un oggetto possiede 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*
- il nome del progetto;
- la 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;
- il *path* dell'icona di *datapack* e *resourcepack*, che verrà prelevata dalle risorse;
- la descrizione in formato JSON o stringa di *datapack* e *resourcepack*, richiesta dal file `pack.mcmeta` di entrambi.
- uno o più *build path*, ovvero cartelle radice in cui saranno esportati il *datapack* e *resourcepack* costruiti. In genere questa coinciderà con la cartella globale di *Minecraft*, nella quale sono raccolte tutte le *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 comporre ogni *path* cui dovrà esportare i file di *datapack* e *resourcepack*.

Un ulteriore *design pattern* applicato a `Project` è *singleton*, che garantisce l'esistenza di un'unica istanza della classe nell'intero programma, accessibile da qualsiasi 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`.

La classe `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`.

Questa implementa le interfacce `Buildable` e `Versionable`, fornendo così i metodi per ottenere i *pack format* corrispettivi alla versione del progetto.
Dispone inoltre di un attributo `namespaces` di tipo `Map` [48], nel quale verranno salvati i `Namespace`. Tramite il metodo `makeMcMeta()` viene generato il file `pack.mcmeta`, obbligatorio per *datapack* e *resourcepack*. Esso comunica a *Minecraft* il valore di `pack_format`, dipendente dalla versione per la quale è stato sviluppato, oltre alla descrizione del *pack*.
Il metodo `build()` è sovrascritto affinché iteri su tutti i valori del dizionario `namespaces`, propagando la costruzione.

Il metodo `addNamespace()`, accennato precedentemente, non aggiunge direttamente il *namespace* al progetto. Prima divide i `FileSystemObject` che contiene tra quelli inerenti alle risorse (*assets*) e quelli relativi alla logica (*data*). Questa suddivisione viene fatta chiamando il metodo polimorfico già 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`.

L'invocazione del metodo `build()` si propaga a cascata partendo da `Project` verso i campi `datapack` e `resourcepack`, i quali delegano l'operazione ai rispettivi `namespace`. Questi a loro volta estendono l'esecuzione a tutti gli elementi figli (cartelle e file), garantendo così il completo attraversamento dell'albero.

Con gli oggetti descritti fino ad ora è possibile costruire un intero *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` dedicati alla localizzazione richiedono un aggiornamento costante per integrare le 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 modificati manualmente, ma generati automaticamente tramite codice Java proprietario.

Proprio perché questo tipo di risorse non è concepito per essere modificato manualmente, sono stati implementati nella classe `Util` metodi dedicati per aggiungere elementi alle risorse in modo programmatico, accessibili da qualunque parte del progetto.

Questo sistema si appoggia ad una funzione che permette di ottenere un riferimento all'oggetto ricercato, o di crearne uno nuovo qualora questo non venga 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 32: Metodo che sfrutta la programmazione funzionale per restituire il `JsonFile` cercato.

Il metodo accetta in input il tipo della classe, il nome dell'oggetto ricercato e un `Supplier` [49]. L'esecuzione avvia uno `Stream` [50] sugli elementi figli del *namespace*, mappando ciascuno di essi tramite l'invocazione di `find()`: tale operazione genera una sequenza di `Optional` che viene filtrata per scartare i risultati vuoti. La pipeline estrae quindi il valore dell'eventuale corrispondenza, restituendo il primo esito utile tramite `findFirst();` avvolgendolo in un `Optional`. Qualora la ricerca non produca alcun esito, e dunque l'`Optional` è vuoto, viene invocato il `Supplier` per generare e restituire una nuova istanza.

Si garantisce così che il metodo restituisca l'oggetto ricercato o ne crei uno nuovo qualora non venga trovato. 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, come si può vedere nel frammento di codice seguente.

```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 33: 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 contenente comandi alla lista di funzioni da eseguire ogni *tick* o ad ogni caricamento dei file.

4.4.2. Ottenimento Versioni

Nel `Builder` di `Project`, in base alla versione di gioco specificata, si ottengono i valori del *pack format* per *datapack* e *resourcepack*. Questi sono memorizzati in un `Record` [54] chiamato `VersionInfo`.

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

1. si controlla che sia presente nel *path* del progetto `resources/_generated` il file `versions.json` contenente tutte le versioni e i format associati;
2. si 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. il file viene letto e convertito in `JsonObject`
5. qualora `versionKey` coincida con `"latest"`, indicando la necessità di recuperare la versione più recente, si istanzia un `Iterator`[7] sulla collezione di `JsonObject`. Il primo elemento estratto viene quindi convertito nel `Record` `VersionInfo`.

6. se `versionKey` corrisponde al nome di una versione, viene restituito l'oggetto `VersionInfo` corrispondente alla chiave richiesta. Questo conterrà i *pack format* richiesti da *datapack* e *resourcepack*.

La generazione di `versions.json` avviene mediante una chiamata HTTP [55] verso un'API [56] dedicata, la quale restituisce un oggetto JSON contenente i dati completi di tutte le versioni disponibili.
Queste vengono poi mappate al nome della versione corrispondente e ordinate dalla più recente alla più vecchia. La mappa così 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 creato le cartelle *datapack* e *resourcepack* di appoggio tramite il metodo `build()`, provvede a comprimerle generando i rispettivi archivi `.zip`. Al termine dell'operazione, queste ultime vengono eliminate.

Il metodo `zipDirectory()` si occupa di comprimere il contenuto di una cartella in un archivio `.zip`. Questo 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 quelli relativi a 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.
Per ogni file trovato, il metodo istanzia una nuova *entry* ZIP, ovvero il contenitore che rappresenta il file all'interno dell'archivio. Per riempirla con i dati effettivi, viene aperto uno `stream` di lettura sul file sorgente: il contenuto viene quindi inserito nell'archivio tramite la classe `IOUtils` [57] 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* [58] 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* [59]. Questi archivi, al fine di evitare confusione tra le versioni, vengono automaticamente nominati con la versione specificata nel file `pom.xml` [60] del progetto Java e saranno scaricabili dalla pagina GitHub che contiene gli artefatti associati alla *release*.

[7] L'utilizzo dell'`Iterator` è indispensabile per accedere al primo elemento, poiché l'interfaccia `Set` non supporta l'accesso posizionale diretto (es. `getFirst()`).

4.5. Implementazione del Working Example

In questa sezione si espone lo sviluppo di un progetto che utilizza la libreria per generare un *pack* che modifica un *item* di *Minecraft*. L'obiettivo è fare in modo che, al clic con il tasto destro del mouse, l'oggetto consuma 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* da utilizzare:

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

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

```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 visivo dell'*item*.

```
Item.f.ofName(id,"""                                                    Java
        {
          "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()` si occupa di creare la *recipe*, ovvero il file JSON che indica gli ingredienti richiesti per creare l'oggetto 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 comunicare tramite un messaggio sullo schermo che la ricetta è disponibile.

Il modulo `MostraRaggio` si occupa di aggiungere comportamenti all'oggetto `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.

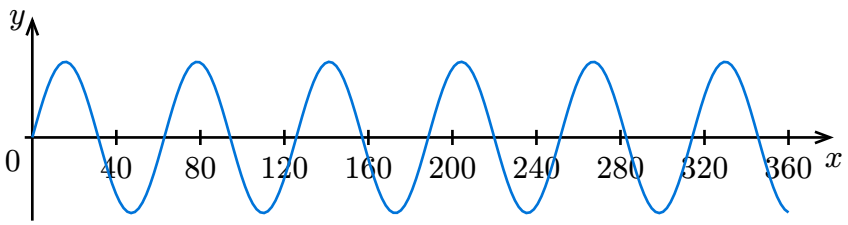

Figura 5: Rappresentazione dei valori memorizzati nella *lookup table*.

[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 le *scoreboard* utili al funzionamento del progetto. La *scoreboard* `click` ha la particolarità di essere automaticamente incrementata ogni volta che il giocatore clicca il tasto destro del mouse, mentre `var` è usata 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 42) che sarà passata come `varargs` della factory, la quale sostituirà `%s`.

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

Codice 41

La funzione di seguito riportata invoca Codice 43 se il giocatore ha cliccato l'*item*, e in seguito azzera il valore della *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 42

I seguenti comandi si occupano di controllare se il giocatore possiede *item* identificati come `ammo`. In caso negativo viene bloccato il flusso di esecuzione, e in caso positivo viene invocata una funzione il cui contenuto è costruito tramite Codice 44, per ottenere la prima munizione che il giocatore possiede. Se è stata trovata una munizione, viene eseguito Codice 45.

```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 43

Il metodo seguente genera i comandi per controllare i 36 *slot* del giocatore. L'esecuzione di quest'ultimi viene arrestata appena viene individuato il primo *item* contrassegnato come `ammo` e memorizzato in uno *storage*.

```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 44

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

1. Codice 45-2: salva la distanza associata alla munizione nello *score* `distance`;
2. Codice 45-3: 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 45-4: chiama una funzione *macro* che elimina la munizione trovata dallo *slot* corrispondente;
4. Codice 45-5: sposta l'esecuzione della funzione all'altezza degli occhi del giocatore, e invoca Codice 46.

```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 45

La seguente funzione rappresenta il nucleo della logica ricorsiva per creare l'onda. Essa decrementa lo *score* `distance`, e memorizza l'esito di questa operazione in uno *storage*. Se ancora non si è raggiunta la distanza massima, ovvero `$ns$.var matches 1..` (var ≥ 1) si sposta l'esecuzione 0.1 blocchi in avanti e si ripete la funzione.
Codice 46-4 invoca la funzione Codice 47, 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 46

Questa funzione contiene un solo comando *macro*, che ne invoca un'altra, passandole il valore corrispondente a $\sin(\texttt{amount} \times 10)$.

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

Codice 47

Questo valore è usato per determinare la posizione verticale della *particle*, relativa al contesto di esecuzione, dando quindi l'impressione che si stia muovendo secondo una funzione sinusoidale.

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

Codice 48

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();
```

È dunque possibile creare una *repository* e pubblicare una *release*. In seguito una *GitHub action* esegue il progetto per generare le due cartelle compresse e rinominarle. In questo caso sono chiamate `datapack-esempio-1.0.0.zip` e `resourcepack-esempio-1.0.0.zip`. Queste sono immediatamente scaricabili e utilizzabili dai giocatori.

Conclusione

Il presente lavoro di tesi ha affrontato le criticità relative allo sviluppo di contenuti per *Minecraft* attraverso il *Domain Specific Language* nativo *mcfunction*. L'analisi preliminare ha rivelato come questo linguaggio, sebbene dotato di *feature* affini a quelle dei linguaggi *general purpose*, imponga severi vincoli strutturali e sintattici. La mancanza di costrutti ad alto livello, combinata con l'obbligo di separare ogni funzione e risorsa in un file distinto, genera codice prolisso, frammentato e difficilmente manutenibile.

Per superare tali limitazioni, è stata progettata e implementata una libreria Java (*OOPACK*) che introduce un approccio orientato agli oggetti per consentire la meta-programmazione di *pack*.

La meta-programmazione è un paradigma che abilita un software di operare su programmi o linguaggi trattandoli come dati. Ciò rende possibile la loro manipolazione o generazione in maniera dinamica o in fase di compilazione [61].

La soluzione proposta astrae la struttura di *datapack* e *resourcepack* in un albero di oggetti tipizzati, consentendo agli sviluppatori di definire molteplici risorse all'interno di un unico contesto e di sfruttare i costrutti di un linguaggio *general purpose*. Attraverso l'automazione della generazione del *boilerplate* e la validazione a tempo di compilazione, il framework riduce drasticamente la complessità di gestione dei file e aumenta la densità di codice, offrendo un ambiente di sviluppo più robusto e scalabile rispetto agli strumenti tradizionali.

Al fine di misurare concretamente l'efficienza della libreria, è stata sviluppata una classe `Metrics` con il compito di registrare il numero di righe e di file generati. Eseguendo il

progetto Java 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 del progetto è invece strutturato nei 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.

Prendendo come riferimento un esempio più articolato, tratto da un progetto personale che mira a inserire nuove piante nel gioco, 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).

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

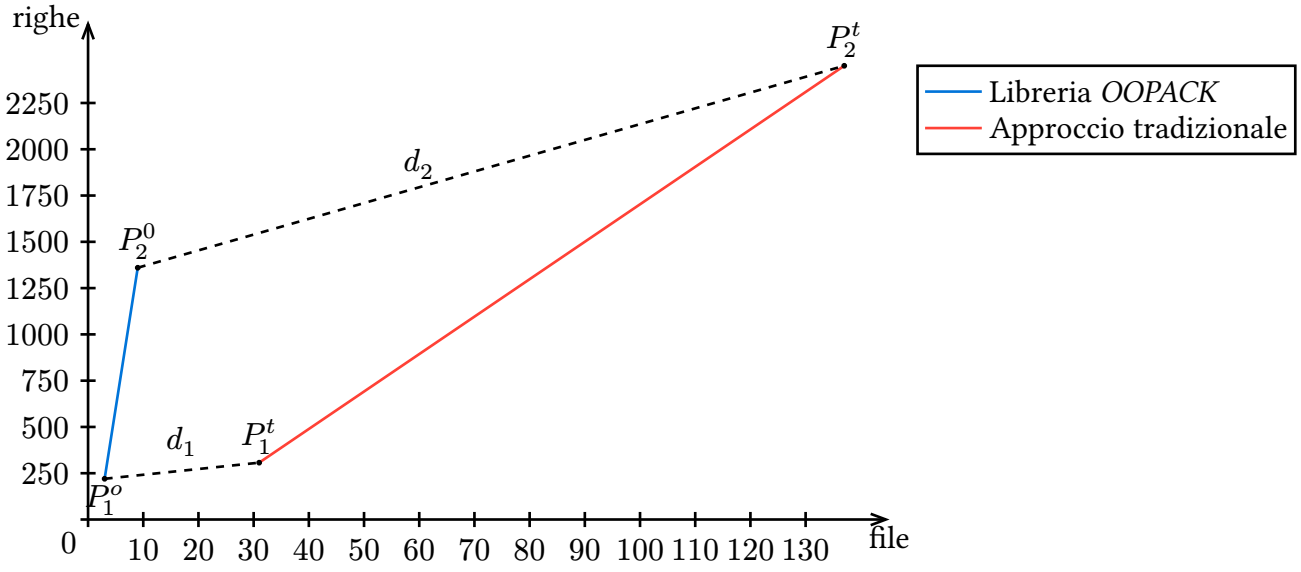

Figura 6: Numero di righe e file richiesti a confronto.

Calcolando il rapporto tra le componenti dell'asse delle ascisse, ovvero il numero dei file impiegati, per P_1 si nota che $\frac{31}{3} = 10,3$: ogni file sorgente genera dunque circa 10,3 file di output. Eseguendo la medesima operazione per P_2 si ottiene invece $\frac{137}{9} = 15,2$. Da questi dati si deduce che la libreria è dotata di «economie di scala positive»: maggiore è la portata del progetto, più elevata è la quantità di file gestita automaticamente per ogni singola unità di codice scritta dallo sviluppatore.

Si può osservare come la linea blu relativa ai progetti sviluppati con la libreria presenti una pendenza maggiore a dimostrazione di un'elevata densità di contenuti per singolo file sorgente.

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 con contenuti affini.

Interpretando la distanza tra il punto di partenza (sorgente) e quello di arrivo (output) come una stima del carico di lavoro automatizzato dalla libreria, è evidente che automatizzare lo sviluppo sia vantaggioso per i progetti di scala maggiore. Per il progetto minore P_1, la distanza è $d_1 = \sqrt{(3-31)^2 + (220-307)^2} = 91,4$. Per il progetto maggiore P_2, tale valore sale a $d_2 = \sqrt{(9-137)^2 + (1360-2451)^2} = 1098,5$.

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

modo significativo all'aumentare della complessità del progetto, ammortizzando rapidamente il costo iniziale di configurazione.

Va tuttavia rilevato che l'utilizzo della libreria richiede un considerevole sforzo cognitivo, dovuto alla necessità di operare simultaneamente con due linguaggi diversi per sfruttare appieno le potenzialità di entrambi.

Si riconosce inoltre la possibilità di estendere la libreria con ulteriori metodi di utilità, potenzialmente più specifici ma comunque in grado di ridurre il carico di lavoro per lo sviluppatore. Per esempio, si potrebbe implementare un metodo che, dati uno o più valori costanti in input, crei la funzione contenente i comandi `scoreboard` con il compito di inizializzare i valori delle costanti *scoreboard*.

Oltre alle conoscenze tecniche acquisite, lo sviluppo del progetto su un arco temporale prolungato ha consentito di rivedere, migliorandole, alcune scelte implementative iniziali. Ciò ha portato ad ottimizzare alcune porzioni di codice che, pur funzionando correttamente, non rappresentavano la soluzione più efficiente né l'approccio più agevole per l'utente finale.
Tale processo di revisione ha favorito lo sviluppo di un'analisi critica della qualità complessiva del software, con 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] «IO Utils». [Online]. Disponibile su: https://commons.apache.org/proper/commons-io/apidocs/org/apache/commons/io/IOUtils.html

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

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

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

[61] «meta-programmazione». [Online]. Disponibile su: https://maddevs.io/glossary/metaprogramming/