|  |  |
| --- | --- |
| Sviluppo di una estensione per VSCode | |
|  | |
| Studente/i | Relatore |
| Beffa Bryan | Coluzzi Massimo |
| Correlatore |
| - |
| Committente |
| - |
| Corso di laurea | Codice progetto |
| Ingegneria informatica | C10652 |
| Anno |  |
| 2022/2023 |  |

|  |  |
| --- | --- |
| Data |  |

|  |  |
| --- | --- |
|  |  |

Indice generale

1 Introduzione 6

2 Stato dell’arte 6

2.1 Extension Pack for Java 6

2.1.1 Language Support for Java 6

3 Motivazione e contesto 8

4 Problema 8

4.1 Obiettivo e requisiti 8

4.1.1 Conoscere ed utilizzare VSCode 8

4.1.2 Comprensione sviluppo estensioni 8

4.1.3 Studio Extension Pack for Java 8

4.1.4 Generazione automatica di codice 9

4.1.5 Creazione e sviluppo estensione 9

5 Implementazione e sviluppo 10

5.1 FileAnalyzer 10

5.1.1 Schema generale 11

5.1.2 Costruttore 12

5.1.3 init() 12

5.1.4 isVisibleField() 12

5.1.5 getClassFields() 13

5.1.6 getParentsVisibleFields() 14

5.1.7 getVisibleFields() 14

5.1.8 main() 15

5.2 Creazione estensione 15

5.2.1 Installazione e configurazione Yeoman 15

5.3 Generazione codice 16

5.3.1 Generazione toString() 16

6 Sitografia 21

7 Allegati 22

Indice delle figure

[Figura 1 - Esempio output - FileAnalyzer 10](#_Toc138192774)

[Figura 2 - schema comunicazione vscode - java 11](#_Toc138192775)

[Figura 3 - Assegnazione campi 11](#_Toc138192776)

[Figura 4 - getClassFields() - logica 13](#_Toc138192777)

[Figura 5 - getParentsVisibleFields () - logica 14](#_Toc138192778)

[Figura 6 - main() - logica 15](#_Toc138192779)

[Figura 7 - Yeoman - creazione estensione 16](#_Toc138192780)

[Figura 8 - Generazione toString() - selezione attributi 16](#_Toc138192781)

[Figura 9 - Generazione toString() - scelta layout 17](#_Toc138192782)

[Figura 10 - Printer snippet - implementazione 17](#_Toc138192783)

[Figura 11 - Printer snippet - package.json configuration 18](#_Toc138192784)

[Figura 12 - Printer snippet - suggerimento 18](#_Toc138192785)

[Figura 13 - Printer snippet - codice esempio 19](#_Toc138192786)

[Figura 14 - Printer snipper - risultato esempio 19](#_Toc138192787)

[Figura 15 - toString() - messaggi di errore 20](#_Toc138192788)

Indice delle tabelle (opzionale, in caso di molte tabelle)

**Riassunto / Abstract**

In versione italiana e inglese, max. 1 pagina per versione, senza figure.

Il testo inglese va eventualmente fatto rivedere da un esperto di lingua inglese.

Il riassunto deve dare un’informazione sintetica sul progetto, cioè dire in cosa consiste e quanto è stato realizzato, in modo che il lettore possa farsene rapidamente un’idea e decidere quindi se sia il caso di approfondire l’argomento nelle pagine successive.

È quindi **importante**:

* Catturare l’attenzione del lettore con un testo di sintesi efficace.
* Tralasciare aspetti di cronistoria o commenti sullo svolgimento del lavoro, che possono eventualmente essere inseriti in un rapporto separato.

1. Introduzione
2. Stato dell’arte

## Extension Pack for Java

L'extension pack for Java di Visual Studio Code è un pacchetto, ideato ed implementato direttamente dal team di sviluppo di Microsoft, che include diverse estensioni per il supporto completo dello sviluppo Java in VSCode. Come ogni altra estensione il suo scopo principale è quello di facilitare e rendere più efficiente la fase di stesura del codice risparmiando tempo nelle parte di codice più macchinose così da permettere al programmatore di concentrarsi sulle parti di logica.

Installando *l'Extension Pack for Java*, vengono installate anche altre estensioni:

* Language Support for Java, *RedHat*
* Debugger for Java, *Microsoft*
* Maven Support, *Microsoft*
* Test Runner for Java, *Microsoft*
* Project Manager for Java, *Microsoft*
* Visual Studio IntelliCode, *Microsoft*

### Language Support for Java

L'estensione *Language Support for Java*" per Visual Studio Code è una delle estensioni più popolari per lo sviluppo Java all'interno dell’editor Visual Studio Code. Fornisce una serie di funzionalità avanzate per semplificare il processo di sviluppo Java e migliorare la produttività degli sviluppatori. Mette a disposizione molte funzionalità e tra le principali è possibile trovare:

#### IntelliSense avanzato

L'estensione fornisce un IntelliSense avanzato che offre suggerimenti intelligenti durante la scrittura del codice Java. È possibile ottenere completamenti automatici per le parole chiave, i nomi delle classi, i metodi e le variabili, facilitando la scrittura del codice senza errori.

#### Debugging potente

Supporto per il debugging di applicazioni Java direttamente in Visual Studio Code. È possibile interrompere, eseguire passo-passo il codice, esaminare variabili e controllare lo stack delle chiamate per individuare e risolvere i bug.

#### Gestione delle dipendenze avanzata:

L'estensione offre una gestione delle dipendenze semplificata per i progetti Java. Vi è la possibilità di gestire facilmente progetti esistenti che utilizzano strumenti come Maven e Gradle, risolvere le dipendenze necessarie e gestire i file di configurazione in modo intuitivo.

#### Formattazione del codice configurabile

L'estensione permette di formattare il codice Java in modo coerente e leggibile. Configurazione delle regole di formattazione secondo gli standard di codifica preferiti o utilizzo di uno stile predefinito per mantenere una formattazione uniforme all'interno dei vari progetti.

#### Refactoring avanzato

Grande quantità di operazioni di refactoring che consentono di ristrutturare e ottimizzare il codice Java in modo sicuro e affidabile. Possibilità di eseguire azioni come la rinominazione di variabili, l'estrazione di metodi, il rilevamento e la risoluzione dei duplicati, semplificando il processo di miglioramento della qualità del codice.

#### Generazione di codice automatica

L'estensione offre gli strumenti per generare automaticamente del codice Java di base. Rapida generazione di costruttori, *getter/setter*, metodi *equals*() e *hashCode*() e *toString*(). Permette di risparmiare tempo e riduce la quantità di codice ripetitivo che devi scrivere manualmente.

#### Supporto completo per i test unitari

Supporto completo per test unitari in Java con framework come JUnit. Il programmatore ha la possibilità di eseguire e analizzare i risultati dei test direttamente in Visual Studio Code, semplificando il processo di sviluppo basato sui test.

#### Integrazione con Git

Supporta l'integrazione con Git, consentendo di gestire facilmente le modifiche del codice, eseguire commit e operazioni di controllo di versione direttamente dall'editor.

È inoltre doveroso e necessario specificare che tutte le funzionalità, quelle elencate e descritte in precedenza e quelle non citate, sono altamente configurabili e personalizzabili a discrezione dell’utente permettendo agli utilizzatori di rendere l’ambiente di sviluppo il più efficiente e familiare possibile.

1. Motivazione e contesto
2. Problema

Il problema da risolvere in questo progetto è la creazione di un'estensione per Visual Studio Code che consenta la generazione automatica di codice Java. L'obiettivo è fornire agli sviluppatori uno strumento che semplifichi e acceleri il processo di generazione di codice ripetitivo, consentendo loro di concentrarsi su compiti più complessi e di valore aggiunto durante lo sviluppo delle applicazioni Java.

L'estensione deve essere in grado di analizzare il codice sorgente di una classe Java e individuare tutti i campi visibili alla classe, compresi quelli ereditati. L'utente dovrebbe essere in grado di selezionare un sottoinsieme di campi o tutti i campi stessi, in base alle proprie esigenze. L'estensione deve quindi dare la possibilità di generare una versione alternativa dei metodi *toString()*, *hashCode*() e *equals*() basata sui campi selezionati. Inoltre, dovrà essere in grado di generare i metodi setter con la convenzione "with" per favorire il fluent programming.

## Obiettivo e requisiti

Questo progetto ha diversi obiettivi e mirano a sviluppare una conoscenza approfondita di Visual Studio Code (VSCode), dello sviluppo di estensioni e delle tecniche di generazione automatica di codice, nonché la capacità di creare una propria estensione per la generazione automatica di codice.

### Conoscere ed utilizzare VSCode

L'obiettivo è di acquisire familiarità con l’editor Visual Studio Code, compresi i suoi strumenti, le funzionalità e le capacità offerte. È importante imparare a utilizzare l'editor, a configurare le impostazioni, ad esplorare le estensioni disponibili e a sfruttare al meglio le funzionalità di VSCode durante lo sviluppo per capirne al meglio le potenzialità offerte.

### Comprensione sviluppo estensioni

Comprendere come creare e sviluppare estensioni per VSCode. Un obiettivo chiave è imparare come creare estensioni per Visual Studio Code. Ciò richiede la comprensione delle API di estensione di VSCode, l'uso dei linguaggi e degli strumenti di sviluppo appropriati, e la capacità di integrare e ampliare le funzionalità di base di VSCode per soddisfare le esigenze specifiche del progetto.

### Studio Extension Pack for Java

Comprendere il funzionamento del Extension Pack for Java: L'obiettivo è comprendere in modo approfondito il funzionamento dell'Extension Pack for Java, un pacchetto di estensioni specifico per lo sviluppo Java in VSCode. È importante studiare le funzionalità, i componenti inclusi, le configurazioni disponibili e come utilizzarlo per supportare lo sviluppo Java in modo efficace.

### Generazione automatica di codice

L'obiettivo è di acquisire conoscenze sulle diverse tecniche utilizzate per la generazione automatica di codice. Ciò include la comprensione dei modelli di generazione, degli strumenti e dei framework utilizzati per automatizzare il processo di creazione di codice ripetitivo o standardizzato, e la consapevolezza della/e best practice per la generazione automatica di codice.

### Creazione e sviluppo estensione

Creare un’estensione per la generazione automatica di codice: L'obiettivo finale è sviluppare una propria estensione per Visual Studio Code che consenta la generazione automatica di codice Java. È necessario applicare le conoscenze acquisite sulle estensioni, su VSCode, sul funzionamento dell'*Extension Pack for Java* e sulle tecniche di generazione automatica di codice per implementare in modo efficace le funzionalità richieste. Ciò comporterà la scrittura di codice, la configurazione delle impostazioni. In particolare, è necessario sviluppare un’estensione per la libreria *Nerd4J.lang*.

L’estensione nello specifico dovrà permettere di generare:

* Metodo *toString()* con possibilità di selezionare gli attributi
* Metodi *equals()* e *hashCode()* con possibilità di selezionare gli attributi
* Metodi *withField()* con possibilità di selezionare gli attributi

1. Implementazione e sviluppo

## FileAnalyzer

Come descritto nei requisiti del progetto, l’estensione di vscode deve essere in grado di generare dinamicamente i metodi *toString(), equals(), hashCode()* e *withField()* suggerendo e richiedendo all’utente di selezionare gli attributi che desidera includere.

Per ottenere quindi i campi della classe e gli eventuali campi ereditati è stato utilizzato l’approccio della reflection di java. In particolare, è stata sviluppata una classe ed il conseguente metodo *main* utili ad effettuare l’analisi di una classe passata come parametro. Per il corretto funzionamento del programma l’utente deve passare come argomenti i seguenti parametri:

* **classPath:** percorso della carte che contiene i file compilati (package escluso)
* **className:** nome della classe (compreso il package)
* **isEditable:** indica se il programma deve ritornare solo i campi modificabili (generazione metodi *withField()*)

Esempio di utilizzo:

In questo esempio la classe FileAnalyzer ritorna come output i campi visibili alla classe *com.mvnproject.Car* allocata nel percorso *c:\Users\project\target\classes.*

|  |
| --- |
| java FileAnalyzer c:\Users\project\target\classes com.mvnproject.Car true |

Output:

Il programma stampa una lista di stringhe contenente le seguenti informazioni:

* **Nome della classe:** (indice 0): utile per la generazione dei metodi *withField()*
* Campi della classe
* Campi visibili ereditati



Figura 1 - Esempio output – FileAnalyzer

### Compatibilità

La classe FileAnalyzer è stata compilate con JDK 11. Ciò significa che se l’utente sta utilizzando una jdk di versione più aggiornata verrà sollevata una UnsupportedClassVersionError. Per risolvere questo problema è necessario ricompilare la classe con la versione di JDK in uso.

### Schema generale

Il flusso di funzionamento del sistema è molto semplice e può essere paragonato ad un’architettura client – server. L’estensione di vscode si occupa di effettuare la richiesta, ovvero avviare il programma con il commando illustrato nello schema tramite la libreria *child\_process*, e attende che venga restituito il risultato sullo standard output.

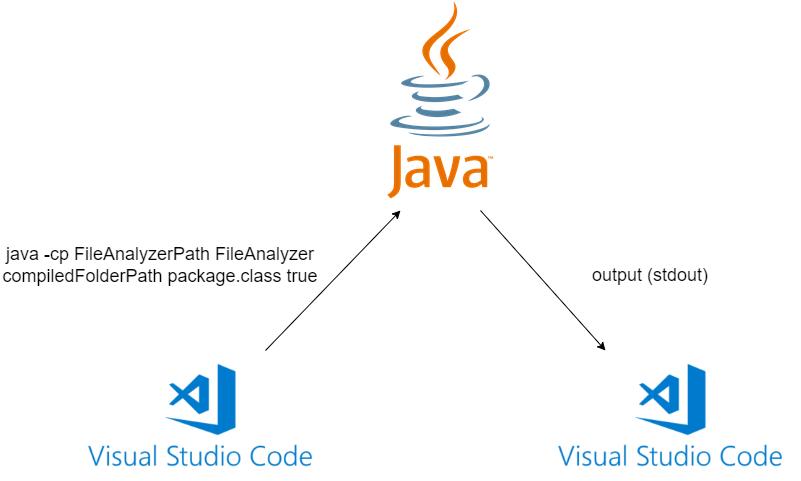


Figura 2 - schema comunicazione vscode - java

Dopo aver ricevuto l'output generato dal **FileAnalyzer**, l'estensione procede a riempire una variabile di tipo vscode.*QuickPickItem[]*. Questa variabile viene utilizzata per visualizzare i campi tramite un popup e permettere la selezione degli elementi.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 3 - Assegnazione campi

### Costruttore

Descrizione:

Il costruttore si occupa di assegnare il valore alla variabile *editableFields* ed effettua una chiamata al metodo *init()*. Questo metodo potrebbe sollevare un’eccezione sollevata dal metodo di inizializzazione.

Parametri richiesti:

* **compiledFilesPath:** Il percorso della directory contenente i file compilati
* **className:** Il nome della classe da analizzare
* **editableFields:** Un valore booleano che indica se includere o meno i campi modificabili

Esempio di utilizzo:

|  |
| --- |
| fileAnalyzer = new FileAnalyzer(compiledFilesPath, className, editableFields); |

### init()

Descrizione:

Il metodo *init()* è un metodo privato che viene utilizzato per inizializzare la classe caricando il file compilato e impostando la variabile *loadedClass* con la classe corrispondente qualora il percorso specificato esista, altrimenti solleva un’eccezione *NoSuchFileException*.

Parametri richiesti:

* **compiledFilesPath**: il percorso della directory contenente i file compilati
* **className**: il nome della classe da caricare.

### isVisibleField()

Descrizione:

Questo metodo privato verifica se un campo specificato è visibile o meno alla classe. Restituisce un valore booleano che indica se il campo è visibile o meno. Questo metodo è utilizzato in particolare per controllare i campi ereditati dalle superclassi. Un campo è considerato secondo i seguenti criteri:

* pubblico, protected
* non è statico

Parametri richiesti:

* **field**: L'oggetto di tipo *Field* che rappresenta il campo da verificare

Valore di ritorno:

* **true:** se il campo è visibile
* **false:** se il campo non è visibile

### getClassFields()

Descrizione:

Il metodo *getClassFields()* restituisce una lista di tutti i campi della classe corrente. Vengono esclusi i campi statici e, se la variabile *editableFields* è settata a **true**, i campi di tipo *final.* Viene riempita e ritornata una lista di stringhe che contiene per ogni campo il tipo di dato ed il nome.

Parti importanti del codice:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 4 - getClassFields() - logica

Valore di ritorno:

Una lista di stringhe che rappresentano i campi della classe

Esempio di utilizzo:

|  |
| --- |
| List<String> visibleFields = fileAnalyzer.getClassFields(); |

### getParentsVisibleFields()

Descrizione:

Questo metodo restituisce una lista di tutti i campi ereditati dalle superclassi della classe caricata. Questi campi sono considerati visibili se soddisfano i criteri definiti dal metodo *isVisibleField()* e non sono dichiarati come private. Viene ritornata una lista di stringhe che contiene per ogni campo il tipo di dato ed il nome.

Parti importanti del codice:

Immagine che contiene testo, schermata

Descrizione generata automaticamente

Figura 5 - getParentsVisibleFields () - logica

Valore di ritorno:

Una lista di stringhe che rappresentano i campi visibili delle superclassi ereditati dalla classe corrente.

Esempio di utilizzo:

|  |
| --- |
| List<String> visibleFields = fileAnalyzer.getParentsVisibleFields(); |

### getVisibleFields()

Descrizione:

Questo metodo restituisce una lista di tutti i campi visibili della classe caricata. Comprende il nome della classe, i campi della classe corrente ottenuti tramite il metodo *getClassFields()*, e i campi visibili ereditati dalle superclassi ottenuti tramite il metodo *getParentsVisibleFields()*.

Valore di ritorno:

Lista di stringhe contenente nome, campi della classe e campi visibili ereditati

Esempio di utilizzo:

|  |
| --- |
| List<String> visibleFields = fileAnalyzer.getVisibleFields(); |

### main()

Il programma utilizza la classe *FileAnalyzer* ed i relativi metodi messi a disposizione. Il codice è composto da poche righe in cui viene creata un’istanza dell’Analyzer e, dopo aver effettuato una chiamata al metodo *getVisibleFields(),* scrive sullo standard output i vari elementi della lista.

Immagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamente

Figura 6 - main() – logica

Successivamente l’estensione di vscode si occupa di recuperare i dati inviati sullo standard output e ad assegnare questi valori alle variabili che successivamente vengono utilizzate durante la generazione dei metodi *toString(), equals(), hashCode()* e *withField().*

## Creazione estensione

Per la creazione di un’estensione in VSCode è stato scelto l’utilizzo di *Yoemen*, un sistema di scaffolding generico che consente di creare applicazioni di qualsiasi tipo. Consente inoltre di avviare rapidamente nuovi progetti e semplificare la manutenzione di quelli esistenti. Questo strumento mette a disposizione una grande quantità di plugin per la configurazione iniziale di applicativi. La scelta dell’utilizzo di questo strumento è stata presa in base alla semplicità di installazione, utilizzo ed efficienza. *Yeoman* è inoltre il sistema più utilizzato per l’inizializzazione di estensioni per VSCode.

Il primo passo da seguire è quello di inizializzare il progetto utilizzando il seguente commando, da terminale, nella cartella root del progetto:

|  |
| --- |
| npm init |

In questo modo viene creato un file package.json che verrà utilizzato per la gestione delle dipendenze dell’estensione. A questo punto è possibile configurare, facoltativamente, alcune impostazioni del progetto, come ad esempio la repository di git, il numero della versione o la descrizione del progetto.

### Installazione e configurazione Yeoman

Installazione di Yeoman è molto semplice e non richiede particolari requisiti se non avere installato sulla macchina su cui si sta lavorando Node.js e il gestore di pacchetti npm. Per installare Yeoman ed il relativo generatore di estensioni per VSCode è necessario eseguire nella root del progetto il seguente comando da terminale:

|  |
| --- |
| npm install -g yo generator-code |

A questo punto è possibile utilizzare e creare la base per l’estensione tramite il generatore di estensioni digitando:

|  |
| --- |
| yo code |

Una volta fatto ciò a terminale apparirà un menù dove è necessario selezionare il tipo di estensione che si desidera creare. In questo caso è stata scelta l’opzione *new Extension (Typescript)*.

Immagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamente

Figura 7 - Yeoman - creazione estensione

Sarebbe stato possibile selezionare anche Javascript, in questo progetto però è stato utilizzato typescript per i seguenti principali motivi:

* Tipizzazione statica
* Controllo degli errori di tipo
* Maggiore robustezza del codice
* Scalabilità e manutenibilità migliorata
* Interfacce e classi

## Struttura estensione

La struttura delle cartelle viene creata automaticamente dallo strumento *Yoeman* seguendo quello che è lo standard utilizzato anche da *Microsoft.* Alla struttura di base sono successivamente state aggiunte delle sottodirectory per organizzare al meglio i file ed il codice del progetto. In particolare, verrà analizzata la seguente struttura:

Immagine che contiene diagramma, testo, linea, schermata

Descrizione generata automaticamente

Figura 8 – struttura modificata

### Directory snippets

In questa directory vengono allocati i file basati su JSON con formato .code-snippets. In questi file è possibile dichiarare frammenti di codice che vengono suggeriti durante la fase di sviluppo di un codice. Per ognuno di questi file è possibile scegliere il contesto in cui questi frammenti possono essere suggeriti.

All’interno di questa directory sono presenti due sottodirectory *dependecies* e *java* che contengono relativamente i frammenti di codice relativi alle dipendenze di Apache Maven, Apache Buildr, Apache Ant, Grails, Groovy Grape, Leiningen, SBT ed i frammenti di codice java relativi agli import delle librerie.

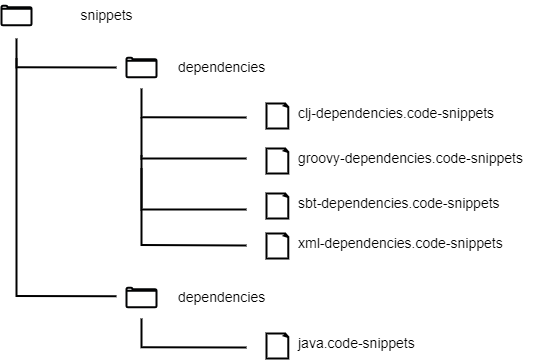


Figura 9 - directory snippets

### Directory src

La directory *src* contiene le directory *java*, *test* e i file typescript, tra cui il punto di accesso dell’estensione *extension.ts.* Nella sottocartella *java* sono allocati i file *FileAnalyzer.java* e *FileAnalyzer.class*, i quali non devono essere spostati per il corretto funzionamento dell’estensione. All’interno della cartella *test* èinvece presente una gerarchia che si occupa di eseguire i test dell’estensione.

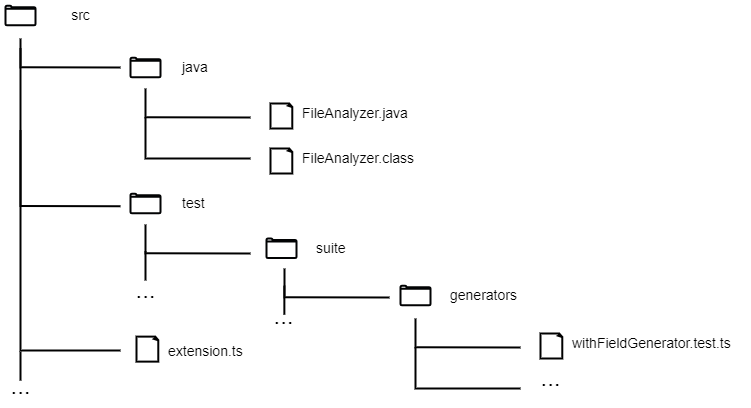


Figura 10 - directory src

### Package.json

Il file *package.json* in un'estensione di VSCode definisce le informazioni e la configurazione dell'estensione stessa. Viene utilizzato per specificare le proprietà, le dipendenze, le attivazioni, i comandi e altre configurazioni come, ad esempio, la definizione dei file utilizzati per gli snippet di codice.

#### Supporto linguaggi

La sezione "languages" nel file *package.json* indica i linguaggi di programmazione supportati dall'estensione e fornisce informazioni su come riconoscere i file associati ad essi. In questo caso, essendo un’estensione sviluppata per la libreria *Nerd4J*, il linguaggio di programmazione scelto è java.

|  |
| --- |
| "languages": [  {  "id": "java",  "extensions": [  ".java"  ]  }  ] |

#### Definizione snippets

Per la definizione degli snippets di codice è necessario inserire all’interno dell’array “snippets” i percorsi dei file ed il linguaggio supportato:

|  |
| --- |
| "snippets": [  {  "language": "java",  "path": "./snippets/java/java.code-snippets"  },  …  ] |

#### Definizione comandi

All’interno del file *package.json* è possibile definire i comandi dell’estensione che possono essere eseguiti tramite la command palette di VSCode oppure tramite una combinazione di tasti. Per ogni comando è necessario specificare il “command” e opzionalmente è possibile scegliere un nome, una descrizione e altre proprietà facoltative.

|  |
| --- |
| "commands": [  {  "command": "madnessjavaextension.showContextMenu",  "title": "Generate ...",  "when": "resourceLangId == 'java'"  },  {  "command": "madnessjavaextension.setCustomCompiledFolder",  "title": "Set custom compiled files folder"  },  {  "command": "madnessjavaextension.deleteCustomCompiledFolder",  "title": "Delete custom compiled files folder"  }  ] |

### Extension.ts

Il file *extension.ts* contiene il punto di ingresso dell'estensione. All’avvio di VSCode, questo file viene caricato e viene eseguita la funzione *activate()*. In questa funzione, vengono eseguite le attività di inizializzazione e configurazione dell'estensione.

Nel file *extension.ts* vi è la possibilità di registrare i comandi personalizzati che successivamente l'estensione stessa metterà a disposizione degli utenti. Questi comandi sono associati a una funzione o un'azione specifica che viene eseguita nel momento in cui il comando viene chiamato tramite comanda palette o una combinazione di tasti.

#### Comandi

Come anticipato all’interno del metodo *activate()* vengono registrati i comandi ed il comportamento che devono assumere nel momento in cui verranno utilizzati. In questi metodi vengono utilizzate funzioni messe a disposizione dal file *codeGenerator.ts*, che si occupa più precisamente di generare il codice richiesto sotto forma di stringa.

*Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente*

Figura 11 - definizione comportamento comandi

I comandi definiti sono:

* **generateToString**: comando per generare il metodo *toString()* in base ai campi selezionati
* **generateWithField**: comando per generare i metodi *withField()* in base ai campi selezionati
* **generateEquals**: comando per generare i metodi equals() e hashCode() in base ai campi selezionati
* **generateAllMethods**: comando per generare *toString()*, *withField(), equals()* e *hashCode()* in base ai campi selezionati
* **setCustomCompiledFolder**: comando per definire un percorso personalizzato della cartella contente i file di java compilati
* **deleteCustomCompiledFolder**: comando per eliminare il percorso personalizzato
* **showContextMenu**: comando per mostrare il menu con le opzioni di generazione di codice

Oltre alla definizione dei vari metodi è buona prassi gestire correttamente le risorse create dall'estensione e garantire una pulizia efficace di tali risorse quando non sono più necessarie. Per fare ciò viene utilizzato un array per tenere traccia di tutte le risorse create dall'estensione in un unico punto.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 12 – subscriptions

#### getFields()

Nel file *extension.ts* è stato aggiunto un metodo di supporto ai comandi che permettono la generazione di codice per ottenere e aggiornare i campi visibili della classe java che si sta modificando.

Descrizione:

Questo metodo utilizza la classe *FileAnalyzer* che, come visto in precedenza, sfrutta la reflection di Java per ottenere i campi visibili di una classe. La funzione *getFields()* recupera il percorso della root del progetto corrente, il percorso completo dei file compilati e il percorso del file attualmente attivo nell'editor di Visual Studio Code al fine di costruire il comando java con i relativi percorsi corretti per eseguire il *main()* della classe *FileAnalyzer*. Successivamente, esegue il comando Java ed in caso di successo viene utilizzato ed elaborato l'output del programma per ottenere l’elenco di campi della classe.

Parametri richiesti:

* **editableField** (opzionale): Un booleano che specifica se i campi devono essere modificabili. Il valore predefinito è false

Valore di ritorno:

Una promise che si risolve in un array di opzioni che rappresentano i campi.

## Generazione codice

In questa sezione è possibile trovare le parti dell’estensione relative alla generazione di codice java. Vengono spiegate le scelte fatte e come sono state implementate all’interno dell’estensione, in particolare, la generazione dei metodi *toString(), equals(),* *hashCode()* ed infine *withField()*.

### Generazione toString()

Immagine che contiene schermata, testo, software, Software multimediale

Descrizione generata automaticamente

Figura 13 - Generazione toString() - selezione attributi

Dopo aver selezionato gli attributi è possibile scegliere il layout di rappresentazione dei dati basandosi sulle opzioni offerte dalla libreria di Nerd4J. Se viene scelto il layout personalizzato *.like()* viene generato del codice di esempio che l’utente può modificare a propria discrezione.

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

Figura 14 - Generazione toString() - scelta layout

#### Rappresentazione dei dati

È possibile decidere il layout della rappresentazione degli attributi selezionati durante la generazione del metodo toString(). Di seguito è riportata una tabella che mostra il nome del metodo e la conseguente rappresentazione grafica al momento del stampa a schermo dei dati.

|  |  |
| --- | --- |
| Nome metodo | Rappresentazione |
| .likeIntellij() | Person{id=1, age=21 } |
| .likeEclipse() | Person [id=1, name=21] |
| .likeFunction() | Person (id:1, name:21) |
| .likeTuple() | Person<id:1, name:21> |
| .like(Printer printer) | - |

#### Snippet Printer

Se l’utente decide di generare il metodo *toString()* con la formattazione personalizzata, ovvero con l’utilizzo del metodo *like(),* ha la possibilità di generare uno snippet di codice di esempio per favorire la creazione di un *Printer* personalizzato, riducendo notevolmente i tempi.

Per la generazione di questo snippet è stata adottata la tecnica di snippet di Visual Studio Code, ovvero un formato di file basato su JSON con estension .code-snippets che permette la creazione di parti di codice che possono essere espanse all’interno dell’editor durante l’implementazione del codice.

Immagine che contiene testo, schermata

Descrizione generata automaticamente

Figura 15 - Printer snippet - implementazione

Successivamente è stato necessario aggiungere nel *package.json*, nell’array di snippets, il file desiderato ed il linguaggio di programmazione a cui è associato.

Immagine che contiene schermata, testo, Carattere

Descrizione generata automaticamente

Figura 16 - Printer snippet - package.json configuration

Per produrre questo codice di supporto è sufficiente digitare *ToString.Printer* oppure *Printer* e selezionare il suggerimento proposto dall’editor.

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

Figura 17 - Printer snippet - suggerimento

Se l’utente decide di utilizzare lo snippet viene creato il seguente codice ed il puntatore viene posizionato all’interno della prima variabile permettendo una personalizzazione più rapida. Premendo *tab* è possibile spostarsi alla variabile successiva. I campi sulla quale il cursore può spostarsi sono:

* separator: indica il carattere o insieme di caratteri che separano gli attributi
* firstDelimeter: carattere o insieme di caratteri che indica l’inizio della stampa dell'attributo/degli attributi
* lastDelimeter: carattere o insieme di caratteri che indica la fine della stampa dell'attributo/degli attributi
* equalityOperator: carattere o insieme di caratteri che precede il valore di un attributo

Una volta settati queste variabile, ed eventualmente modificato a propria discrezione l’implementazione di esempio generata in automatico, il metodo *toString()* è pronto per essere utilizzato e potrebbe assomigliare all'esempio che segue:

Immagine che contiene testo, schermata, software

Descrizione generata automaticamente

Figura 18 - Printer snippet - codice esempio

Producendo, una volta avviato il programma e chiamato il metodo *toString()*, il risultato:

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

Figura 19 - Printer snipper - risultato esempio

#### Controlli generazione toString()

Prima di generare il codice viene controllato che il metodo *toString*()non sia già implementato e se è stato selezionato almeno un attributo della classe. Se uno dei due controlli non viene superato viene mostrato un popup di errore all’utente.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 20 - toString() - messaggi di errore

1. Sitografia

<https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack>, *Extension Pack for Java*, 30.05.2023

<https://marketplace.visualstudio.com/items?itemName=redhat.java>, *Language support for Java,* 30.05.2023

<https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-debug>, *Debugger for Java,* 30.05.2023

<http://nerd4j.org/utils/package/lang.html>, *Libreria Nerd4J,* 30.05.2023

<https://yeoman.io/learning/index.html>, *Yeoman,* 31.05.2023

<https://github.com/redhat-developer/vscode-java>, *Language support for Java - Github repository,* 01.06.2023

<https://github.com/microsoft/vscode-maven>, *Maven for Java – Github repository,* 01.06.2023

1. ****Allegati****

Fanno parte della documentazione, ma non del relativo fascicolo, trattandosi di materiale separato, anche se riferito alla documentazione stessa.

Si tratta del CD contenente la documentazione stessa e altro materiale riferito al progetto, ev. fascicoli separati (come ad es. un Manuale d’uso), ev. materiale sperimentale del progetto.

L’identificazione avviene come per le Appendici, usando però lettere iniziali diverse, in modo da non confondere Appendici e Allegati.

Importante: gli Allegati, essendo separati, vanno sempre corredati di scritte o etichette che li identifichino come relativi al progetto e alla documentazione cui si riferiscono (titolo, codice, ecc.).