|  |  |
| --- | --- |
| Sviluppo di una estensione per VS Code | |
|  | |
| 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 7

2 Stato dell’arte 7

2.1 IDE 7

2.1.1 Tipologie di IDE 8

2.2 Visual Studio Code 8

2.2.1 Linguaggi di programmazione 8

2.2.2 Estensibilità 8

2.2.3 Strumenti di debugging 9

2.2.4 Gestione dei progetti 9

2.3 Extension Pack for Java 9

2.3.1 Language Support for Java 9

3 Motivazione e contesto 11

4 Problema 11

4.1 Obiettivo e requisiti 11

4.1.1 Conoscere ed utilizzare VS Code 11

4.1.2 Comprensione sviluppo estensioni 11

4.1.3 Studio Extension Pack for Java 11

4.1.4 Generazione automatica di codice 12

4.1.5 Creazione e sviluppo estensione 12

5 Implementazione e sviluppo 13

5.1 FileAnalyzer 13

5.1.1 Compatibilità 13

5.1.2 Schema generale 14

5.1.3 Costruttore 15

5.1.4 init() 15

5.1.5 isVisibleField() 15

5.1.6 getClassFields() 16

5.1.7 getParentsVisibleFields() 17

5.1.8 getVisibleFields() 17

5.1.9 main() 18

5.2 Creazione estensione 18

5.2.1 Installazione e configurazione Yeoman 19

5.3 Struttura estensione 20

5.3.1 Directory snippets 20

5.3.2 Directory src 21

5.3.3 Package.json 21

5.3.4 Extension.ts 23

5.4 codeGenerator.ts 26

5.4.1 checkIfMethodAlreadyExists() 26

5.4.2 getPackageName() 27

5.4.3 replaceOldCode() 27

5.4.4 checkJavadocComment() 28

5.4.5 generateToStringCode() 29

5.4.6 generateEquals() 29

5.4.7 generateHashCode() 29

5.4.8 generateWithFields() 30

5.4.9 getIndentation() 30

5.4.10 insertTab() 31

5.5 Generazione codice 31

5.5.1 toString() 32

5.5.2 equals() e hashCode() 38

5.5.3 withField() 40

6 Sitografia 42

7 Allegati 43

Indice delle figure

[Figura 1 - Esempio output – FileAnalyzer 13](#_Toc143590967)

[Figura 2 - schema comunicazione VS Code - java 14](#_Toc143590968)

[Figura 3 - Assegnazione campi 14](#_Toc143590969)

[Figura 4 - getClassFields() - logica 16](#_Toc143590970)

[Figura 5 - getParentsVisibleFields () - logica 17](#_Toc143590971)

[Figura 6 - main() – logica 18](#_Toc143590972)

[Figura 7 - Yeoman - creazione estensione 19](#_Toc143590973)

[Figura 8 – struttura modificata 20](#_Toc143590974)

[Figura 9 - directory snippets 20](#_Toc143590975)

[Figura 10 - directory src 21](#_Toc143590976)

[Figura 11 - definizione comportamento comandi 23](#_Toc143590977)

[Figura 12 – subscriptions 24](#_Toc143590978)

[Figura 13 - costruzione comando java 25](#_Toc143590979)

[Figura 14 Comando generate 26](#_Toc143590980)

[Figura 15 - Generazione toString() - selezione attributi 26](#_Toc143590981)

[Figura 16 - Generazione toString() - scelta layout 27](#_Toc143590982)

[Figura 17 toString() – risultato 27](#_Toc143590983)

[Figura 18 toString() - inserimento codice 28](#_Toc143590984)

[Figura 19 toString() - sostituzione codice 28](#_Toc143590985)

[Figura 20 - Printer snippet - implementazione 29](#_Toc143590986)

[Figura 21 - Printer snippet - package.json configuration 29](#_Toc143590987)

[Figura 22 - Printer snippet - suggerimento 29](#_Toc143590988)

[Figura 23 - Printer snippet - codice esempio 31](#_Toc143590989)

[Figura 24 - Printer snippet - risultato esempio 31](#_Toc143590990)

[Figura 25 Generazione equals() e hashCode() - selezione attributi 32](#_Toc143590991)

[Figura 26 Opzione creazione metodo hashCode 32](#_Toc143590992)

[Figura 27 equals() e hashCode() – risultato 32](#_Toc143590993)

[Figura 28 equals() e hashCode() - inserimento codice 33](#_Toc143590994)

[Figura 29 equals() e hashCode() - regenerate 33](#_Toc143590995)

[Figura 30 equals() e hashCode() – sostituzione codice 33](#_Toc143590996)

[Figura 31 Generazione withField() - selezione attributi 34](#_Toc143590997)

[Figura 32 withField() - risultato 34](#_Toc143590998)

[Figura 33 withField() - inserimento 34](#_Toc143590999)

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

## IDE

Un Ambiente di Sviluppo Integrato, detto anche IDE è uno strumento software che consente agli sviluppatori di scrivere, testare e debuggare il codice in modo efficiente. L'IDE offre una vasta gamma di funzionalità integrate che rendono il processo di sviluppo software più semplice ed efficiente, permettendo agli sviluppatori di concentrarsi maggiormente sulla logica del loro codice anziché sulla configurazione e sulla gestione delle diverse componenti dello sviluppo.

Uno dei componenti principali di un IDE è l'editor di codice, il quale consente agli sviluppatori di scrivere, modificare e formattare il codice in modo leggibile. Spesso, gli editor di codice forniscono funzionalità di evidenziazione della sintassi, completamento automatico del codice e strumenti di navigazione che ne semplificano la stesura del codice.

L’IDE include anche un compilatore o un interprete. Il compilatore traduce il codice sorgente in un linguaggio macchina eseguibile, mentre l'interprete esegue il codice direttamente senza la necessità di una fase di compilazione separata. Questo ambiente di sviluppo facilita l'integrazione di questi strumenti nel flusso di lavoro di sviluppo e semplifica la gestione degli errori di compilazione o interpretazione.

Un altro componente molto importante di un IDE è il debugger. Questo strumento consente agli sviluppatori di eseguire il codice passo dopo passo, permettendo di identificare e risolvere errori e bug nel programma. Il debugger offre funzionalità come l'impostazione di punti di interruzione, l'ispezione delle variabili e l'esecuzione del codice in modalità di debug per un'analisi più approfondita del comportamento del programma. Grazie a questo tool gli sviluppatori sono in grado di analizzare il codice in maniera più rapida ed efficace.

Un altro aspetto molto importante di questi strumenti software è l’inclusione di strumenti di gestione del progetto che facilitano la creazione, l'organizzazione e la gestione dei file e delle risorse del progetto. Questi strumenti consentono agli sviluppatori di visualizzare l'intera struttura del progetto, gestire le dipendenze esterne e integrare librerie di terze parti minimizzando notevolmente la quantità di lavoro dello sviluppatore.

Inoltre, gli IDE offrono funzionalità come il completamento automatico del codice, che suggerisce automaticamente il completamento delle istruzioni e delle variabili mentre si digita. Questo permette di accelerare il processo di scrittura del codice e riduce gli errori di digitazione.

### Tipologie di IDE

Principalmente esistono due tipologie di IDE, quelli che offrono supporto per più linguaggi di programmazione e quelli che sono progettati per un linguaggio specifico.

Gli IDE multi-linguaggio: sono strumenti software che offrono supporto per lo sviluppo di applicazioni in diversi linguaggi di programmazione. In particolare, sono progettati per consentire agli sviluppatori di lavorare con molteplici linguaggi all'interno dello stesso ambiente di sviluppo integrato. Un esempio di IDE multi-language è VS Code, editor di codice sviluppato da Microsoft.

Gli IDE focalizzati su un linguaggio: specifico sono strumenti progettati specificamente per fornire un supporto avanzato e specializzato per uno o più linguaggi di programmazione particolari. Questi IDE sono ottimizzati per offrire un'esperienza di sviluppo più approfondita ed efficiente per il linguaggio specifico, fornendo strumenti avanzati per il completamento del codice, la refactoring, l'analisi statica e molto altro ancora. Un esempio di IDE focalizzato su un linguaggio specifico è IntelliJ IDEA. IntelliJ IDEA è un IDE sviluppato da JetBrains che offre un supporto eccezionale per lo sviluppo di applicazioni Java.

## Visual Studio Code

Visual Studio Code è un editor di codice sorgente gratuito, estensibile e multipiattaforma sviluppato da Microsoft. È ampiamente utilizzato dagli sviluppatori di tutto il mondo grazie alla sua flessibilità, alla vasta gamma di funzionalità e all'ampia disponibilità di estensioni. Grazie alla vasta community di sviluppatori e di estensioni pubblicate sul marketplace, gli utenti possono utilizzare questo editor di codice come un vero e proprio IDE multi-linguaggio.

### Linguaggi di programmazione

VS Code supporta una vasta gamma di linguaggi di programmazione, inclusi JavaScript, TypeScript, Python, Java, C++, C#, Ruby, PHP, Go, Rust e molti altri. L'editor offre funzionalità di evidenziazione della sintassi, completamento automatico del codice, formattazione del codice, refactoring, navigazione intelligente e suggerimenti di codice contestuali. Queste caratteristiche rendono la scrittura del codice più efficiente e aiutano a ridurre gli errori di sintassi ed eventuali parti di codice duplicate.

### Estensibilità

Una delle caratteristiche distintive di VS Code è la sua estensibilità. L'editor offre un vasto ecosistema di estensioni create dalla community, che consentono di personalizzare e facilitare l'esperienza di sviluppo. Le estensioni possono aggiungere nuove funzionalità, supporto per specifici framework o linguaggi di programmazione, strumenti di debugging avanzati, integrazioni con servizi di terze parti e molto altro ancora. È possibile esplorare e installare estensioni direttamente dall'IDE, rendendo facile adattare VS Code alle proprie esigenze. Inoltre, è possibile sviluppare la propria estensione di VS Code e pubblicarla successivamente sul Marketplace.

### Strumenti di debugging

VS Code offre un potente debugger integrato che supporta diversi linguaggi di programmazione. È possibile impostare punti di interruzione, eseguire il codice passo dopo passo, ispezionare variabili e osservare lo stato del programma durante l'esecuzione. I risultati del debug vengono visualizzati in una finestra dedicata, offrendo un'ampia visibilità sul comportamento del codice e facilitando l'individuazione e la risoluzione degli errori.

### Gestione dei progetti

Questo editor di codice fornisce strumenti per la gestione dei progetti, inclusa la possibilità di creare, aprire e salvare progetti in modo semplice. È possibile organizzare i file all'interno di una struttura di cartelle, esplorare i file e le risorse del progetto, eseguire comandi specifici del progetto e integrare sistemi di controllo versione come Git per la gestione del codice.

## 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 VS Code. 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, 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 VS Code

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 VS Code durante lo sviluppo per capirne al meglio le potenzialità offerte.

### Comprensione sviluppo estensioni

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

### Studio 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 VS Code. 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 VS Code, 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 VS Code 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 VS Code 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.

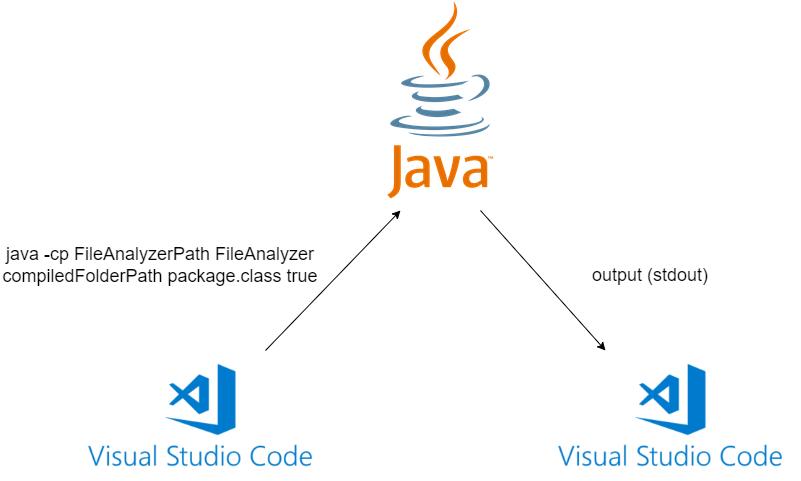


Figura 2 - schema comunicazione VS Code - java

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 VS Code 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 VS Code 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 VS Code.

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 VS Code 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.

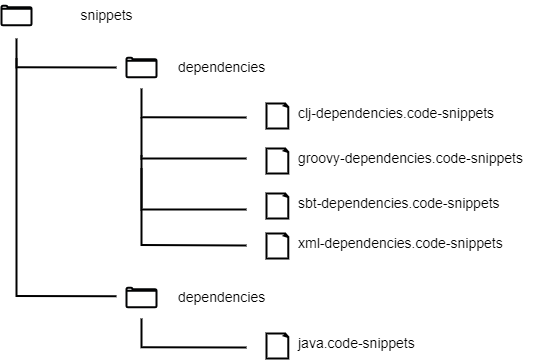


Figura 9 - directory snippets

### 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.

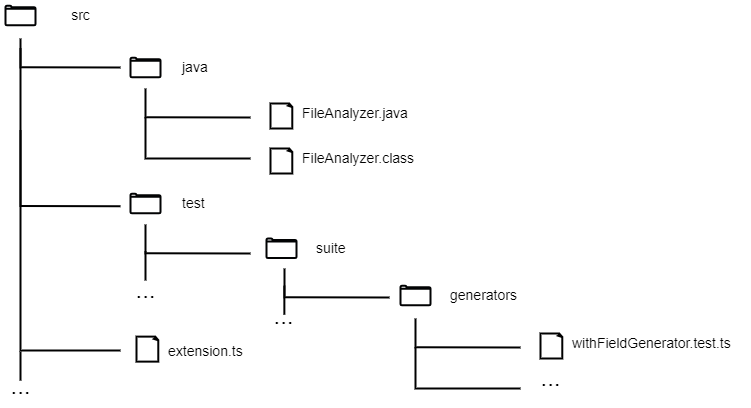


Figura 10 - directory src

### Package.json

Il file *package.json* in un'estensione di VS Code 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 VS Code 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 VS Code, 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
* **setWorkspaceJDK**: permette all’utente di selezionare la cartella della jdk desiderata
* **checkCurrentJDK**: informa l’utente sulla versione corrente della jdk
* **recompileFileAnalyzer**: comando che permette all’utente di ricompilare la classe java

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

Parti importanti del codice:

La parte di codice principale di questo metodo è legato alla costruzione del comando java per avviare il programma *FileAnalyzer*. La difficoltà principale riscontrata è stata quella di costruire i percorsi in maniera corretta, affinché le classi venissero trovate nelle cartelle predefinite oppure nella cartella specificata dall’utente, inserendo tutti i controlli necessari. Inoltre, è stato aggiunto anche il controllo e riconoscimento del package in cui la classe si trova così da poter costruire correttamente il comando java da eseguire.

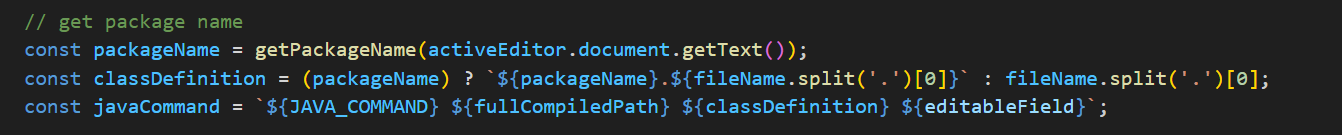


Figura 13 - costruzione comando java

Valore di ritorno:

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

## config.ts

Il file *config.ts* contiene una serie di costanti che sono utilizzate per la configurazione e la definizione di variabili legate all'analisi dei file Java. Il file contiene le seguenti costanti:

* **JAVA\_ANALYZER\_FOLDER**: Questa costante rappresenta il percorso alla cartella contenente il FileAnalyzer per l'analisi dei file Java.
* **JAVA\_FILE\_NAME**: questa costante contiene il nome del file per l’analisi delle classi java
* **JAVA\_COMMAND**: questa costante permette di creare il comando java da eseguire sfruttando le altre due costanti viste in precedenza.
* **JAVAC\_COMMAND**: questa costante ha lo stesso comportamento della precedenza con la differenza che definisce il comando javac, ovvero per la compilazione.
* **TO\_STRING\_IMPORT**: definisce l’import di nerd4j per la classe ToString. In particolare, contiene il valore ‘import org.nerd4j.utils.lang.ToString;’
* **HASHCODE\_IMPORT**: definisce l’import di nerd4j per la classe Hashcode. In particolare, contiene il valore ‘import org.nerd4j.utils.lang.Hashcode;’
* **EQUALS\_IMPORT**: definisce l’import di nerd4j per la classe Equals. In particolare, contiene il valore ‘import org.nerd4j.utils.lang.Equals;’
* **TO\_STRING\_SIGNATURE**: questa costante contiene la signature del metodo *toString()*. Il valore assegnato è ‘public String toString()’
* **EQUALS\_SIGNATURE**: questa costante contiene la signature del metodo *equals()*. Il valore assegnato è ‘public boolean equals(Object other)’
* **HASHCODE\_SIGNATURE**: questa costante contiene la signature del metodo *hashCode()*. Il valore assegnato è ‘public int hashCode()’

## jdkManagement.ts

Questo file viene utilizzato per la gestione della jdk di un progetto java, in particolare mette a disposizione il metodo per verificare se è presente una jdk ed il metodo per configurarla.

Oltre a questi metodi, questo file mette a disposizione una costante *jdkQuickFix* che permette di suggerire ed eventualmente eseguire il comando per settare la jdk in caso di errore durante la generazione dei metodi *toString(), equals()* e *hashCode()*.



Figura 14 jdkManagement – jdkQuickFix

### getCurrentJDK()

Descrizione:

Metodo che ritorna, se è configurata, la jdk corrente utilizzata dal progetto dell’utente. Questo metodo va a controllare se nel file *settings.json* di VSCode è presente la chiave “java.jdt.ls.java.home” e in caso affermativo ne ritorna il valore.

Valore di ritorno:

Stringa contenente il percorso della jdk

Esempio di utilizzo:

|  |
| --- |
| const jdk = getCurrentJDK(); |

### setWorkspaceJDK()

Descrizione:

Questo metodo permette all’utente di aggiornare o inserire all’interno del file *settings.json* di VSCode la chiave ‘java.jdt.ls.java.home’ ed il percorso della jdk selezionato.

Parametri richiesti:

* **jdkPath**: percorso della cartella principale della jdk che si desidera utilizzare

Esempio di utilizzo:

|  |
| --- |
| setWorkspaceJDK(jdkMainFolder); |

## codeGenerator.ts

Il file codeGenerator.ts contiene tutti i metodi utili alla generazione di codice dei metodi *toString(), equals()* e *hashCode()*.

Questo file contiene e mette a disposizione i seguenti metodi:

* checkIfMethodAlreadyExists(methodSignature: string)
* getPackageName(text: string)
* replaceOldCode(regex: RegExp, newCode: string)
* checkJavadocComment(oldCodeIndex: number)
* generateToStringCode(selectedAttributes: string[], selectedType: string)
* generateEquals(selectedAttributes: string[])
* generateHashCode(selectedAttributes: string[])
* generateWithFields(selectedAttributes: string[], className: string)
* getIndentation()
* insertTab(times: number)

### checkIfMethodAlreadyExists()

Descrizione:

Funzione che ritorna se un metodo è presente o meno all’interno del codice, ricevendo come parametro la sua signature.

Parametri richiesti:

* **methodSignature:** la signature del metodo da verificare

Valore di ritorno:

Valore booleano che indica se il metodo è presente all’interno del codice

Esempio di utilizzo:

|  |
| --- |
| if (checkIfMethodAlreadyExists(TO\_STRING\_SIGNATURE)) {  …  } |

### getPackageName()

Descrizione:

Metodo che ritorno, se è presente, la stringa contenente il package completo della classe java su cui l’utente sta effettuando le modifiche.

Parametri richiesti:

* **text**: il codice del file su cui si stanno effettuando le modifiche.

Valore di ritorno:

Stringa che contiene, se presente, il package della classe.

Esempio di utilizzo:

|  |
| --- |
| const packageName = getPackageName(activeEditor.document.getText()); |

### replaceOldCode()

Descrizione:

Metodo che sostituisce il vecchio codice con il codice che gli viene fornito come parametro.

Parametri richiesti:

* **regex**: regular expression che indica il codice del metodo da sostituire
* **newCode**: codice che rimpiazza il codice da sostituire

Parti importanti del codice:

Una volta verificato che all’interno del codice è presente il metodo da sostituire e rigenerare viene calcolato il range utilizzando la posizione iniziale e la posizione finale del codice da rimuovere. Successivamente tramite il metodo *replace()* dell’*editBuilder* è possibile sostituire il range del codice calcolato in precedenza con il nuovo codice generato.

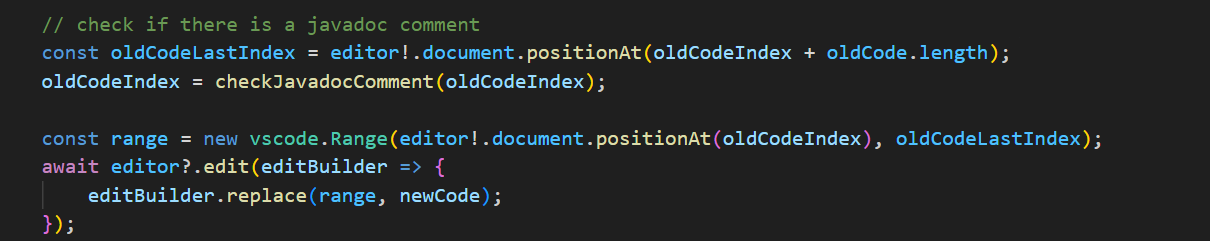


Figura 15 replaceOldCode() - logica

Esempio di codice:

|  |
| --- |
| await replaceOldCode(toStringRegExp, toStringCode); |

### checkJavadocComment()

Descrizione:

Metodo che controlla se è presente un commento javadoc relativo al metodo che si desidera sostituire. Se è presente ritorna l’indice della posizione all’interno del codice dell’editor. In questo modo, se è presente, è possibile rimuovere il metodo da sostituire con il relativo commento javadoc.

Parametri richiesti:

* **oldCodeIndex**: posizione all’interno del codice dell’editor del metodo da sostituire.

Valore di ritorno:

Posizione iniziale del codice da rimuovere all’interno dell’editor.

Parti importanti del codice:

Nella parte principale del codice vengono controllati i caratteri precedenti alla posizione in cui inizia il metodo da sostituire. Per ogni carattere viene controllato se corrisponde alla chiusura di un commento javadoc o all’apertura. Se non è presente un commento il codice si ferma appena incontra il carattere ”;”, “}” oppure “{”.

Immagine che contiene testo, schermata

Descrizione generata automaticamente

Figura 16 checkJavadocComment - logica

Esempio di utilizzo:

|  |
| --- |
| // check if there is a javadoc comment  const oldCodeLastIndex = editor!.document.positionAt(oldCodeIndex + oldCode.length);  oldCodeIndex = checkJavadocComment(oldCodeIndex); |

### generateToStringCode()

Descrizione:

Questo metodo costruisce e ritorna sottoforma di stringa il metodo *toString()* in base ai parametri di entrata.

Parametri richiesti:

* **selectedAttributes**: attributi selezionati dall’utente per la generazione del metodo *toString()*
* **selectedType**: tipologia del layout di rappresentazione dei dati richiesto dall’utente

Valore di ritorno:

Stringa contenente il metodo *toString()*.

Esempio di utilizzo:

|  |
| --- |
| const toStringCode = await generateToStringCode(selectedAttributes, selectionType); |

### generateEquals()

Descrizione:

Questo metodo costruisce e ritorna sottoforma di stringa il metodo *equals()* in base agli attributi selezionati dall’utente.

Parametri richiesti:

* **selectedAttributes**: attributi selezionati dall’utente per la generazione del metodo e*quals()*

Valore di ritorno:

Stringa contenente il metodo *equals()*.

Esempio di utilizzo:

|  |
| --- |
| const equalsCode = await generateEquals(selectedAttributes); |

### generateHashCode()

Descrizione:

Questo metodo costruisce e ritorna sottoforma di stringa il metodo *hashCode()* in base agli attributi selezionati dall’utente.

Parametri richiesti:

* **selectedAttributes**: attributi selezionati dall’utente per la generazione del metodo *hashCode()*

Valore di ritorno:

Stringa contenente il metodo *hashCode()*

Esempio di utilizzo:

|  |
| --- |
| const equalsCode = await generateEquals(selectedAttributes); |

### generateWithFields()

Descrizione:

Questo metodo ritorna una stringa contenente tutti i metodi *withField()* generati in base agli attributi selezionati dall’utente.

Parametri richiesti:

* **selectedAttributes**: attributi selezionati dall’utente per la generazione dei metodi *withField()*

Valore di ritorno:

Stringa contenente i metodi *withField()* che sono stati generati.

Esempio di utilizzo:

|  |
| --- |
| const withFieldCode = generateWithFields(selectedAttributes, className); |

### getIndentation()

Descrizione:

Metodo che ritorna il numero di tab necessari ad indentare correttamente i metodi generati.

Valore di ritorno:

Numero di tab necessari ad indentare correttamente i metodi generati dall’estensione.

Parti importanti del codice:

La parte principale del metodo si occupa di contare il numero di parentesi graffe che sono state aperte, al fine di ritornare la quantità di tab necessaria ad indentare correttamente il codice.

Immagine che contiene testo, Software multimediale, software, Carattere

Descrizione generata automaticamente

Figura 17 getIndentation() - logica

Esempio di utilizzo:

|  |
| --- |
| const tabs = insertTab(getIndentation()); |

### insertTab()

Descrizione:

Metodo che ritorna una stringa che contiene il numero di carattere ’\t’ necessario ad indentare correttamente il codice generato.

Parametri richiesti:

* **times**: numero di ‘\t’ richiesto

Valore di ritorno

Stringa che contiene il numero di carattere ’\t’ necessario ad indentare il codice.

Esempio di utilizzo:

|  |
| --- |
| const tabs = insertTab(getIndentation()); |

## 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()*.

I metodi di generazione del codice sono eseguibili tramite command palette di VSCode digitando il comando “*Nerd4J: generate*”. A questo punto è possibile scegliere tra 4 diverse opzioni di generazione del codice.

Immagine che contiene schermata, testo, software, Software multimediale

Descrizione generata automaticamente

Figura 18 Comando generate

### toString()

Per la generazione del metodo toString() è necessario scegliere “*toString()”* tra le varie opzioni consigliate dall’editor di testo.

Immagine che contiene schermata, testo, software, Software multimediale

Descrizione generata automaticamente

Figura 19 - 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 20 - 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) | - |

#### Risultato

A questo punto il codice del metodo *toString()* che è stato generato viene inserito e indentato correttamente all’interno del file java su cui l’utente sta programmando.

Immagine che contiene testo, schermata

Descrizione generata automaticamente

Figura 21 toString() – risultato

#### Generazione codice

Per generare il metodo *toString()* vengono unicamente passati al metodo *generateToString()* l’array contenente gli attributi selezionati e il layout di rappresentazione dei dati desiderato dall’utente. Una volta costruito il codice viene inserito nel file aggiungendo, se non presente, l’import necessario.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 22 toString() - inserimento codice

#### Rigenerazione codice

Se in precedenza è già stato generato il metodo *toString()* viene chiesto esplicitamente all’utente se desidera rigenerare il codice. In caso di risposta affermativa tramite il metodo *replaceOldCode()* messo a disposizione dal *codeGenerator.ts* viene sostituito il veccho codice con quello appena generato.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 23 toString() - sostituzione codice

#### 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 24 - 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 25 - 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 26 - 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 27 - 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 28 - Printer snippet - risultato esempio

### equals() e hashCode()

Per la generazione dei metodi equals() e hashCode(), come nella generazione del metodo toString(), è necessario eseguire il comando “*Nerd4J: generate*” e successivamente selezionare l’opzione “*equals() and hashCode()*”.

Una volta fatto ciò viene richiesto all’utente di scegliere gli attributi per la generazione del codice dei due metodi tramite il metodo *getFields()*.

Immagine che contiene schermata, testo, software, Software multimediale

Descrizione generata automaticamente

Figura 29 Generazione equals() e hashCode() - selezione attributi

Successivamente, come avviene in diversi ambienti di sviluppo o estensioni, viene richiesto all’utente se desidera generare anche il metodo *hashCode()*. Se viene selezionata l’opzione per la generazione di questo metodo verranno presi in considerazione gli attributi scelti in precedenza all’avvio del comando.

Immagine che contiene schermata, testo, software, Software multimediale

Descrizione generata automaticamente

Figura 30 Opzione creazione metodo hashCode

#### Risultato

A questo punto il codice del metodo *equals(), ed eventualmente hashCode(),* che è stato generato viene inserito e indentato correttamente all’interno del file.

Immagine che contiene testo, schermata, software

Descrizione generata automaticamente

Figura 31 equals() e hashCode() – risultato

#### Generazione codice

La parte di generazione del codice è molto semplice e simile a quella del metodo *toString()*. Per generare i due metodi è richiesto unicamente di passare come parametro l’array contenente gli attributi desiderati dall’utente e, una volta ottenuto il codice, andarlo ad inserire all’interno del file di java nel modo seguente, controllando inoltre che siano presenti tutti gli import necessari:

Immagine che contiene testo, schermata, software, Carattere

Descrizione generata automaticamente

Figura 32 equals() e hashCode() - inserimento codice

#### Rigenerazione codice

Se i metodi richiesti sono già presenti all’interno del codice, l’estensione mostra un popup di informazione che dà la possibilità di generare e sostituire i metodi esistenti oppure annullare la generazione del nuovo codice.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 33 equals() e hashCode() - regenerate

Se l’utente desidera rigenerare i metodi, tramite il metodo *replaceOldCode()* è possibile sostituire i vecchi metodi *equals()* e *hashCode()* con il nuovo codice generato.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 34 equals() e hashCode() – sostituzione codice

### withField()

Per la generazione dei metodi withField è necessario eseguire nuovamente il commando “*Nerd4J: generate*” e scegliere l’opzione “*withField()*”.

La generazione dei metodi withField presenta inoltre una piccola limitazione, ovvero che all’utente sia data unicamente la possibilità di selezionare gli attributi “modificabili” non suggerendo quindi campi *private* che sono stati ereditati o *final*.

Sempre utilizzando il metodo *getFields()* ma specificando come parametro il valore booleano true, è possibile ottenere la lista dei campi modificabili.

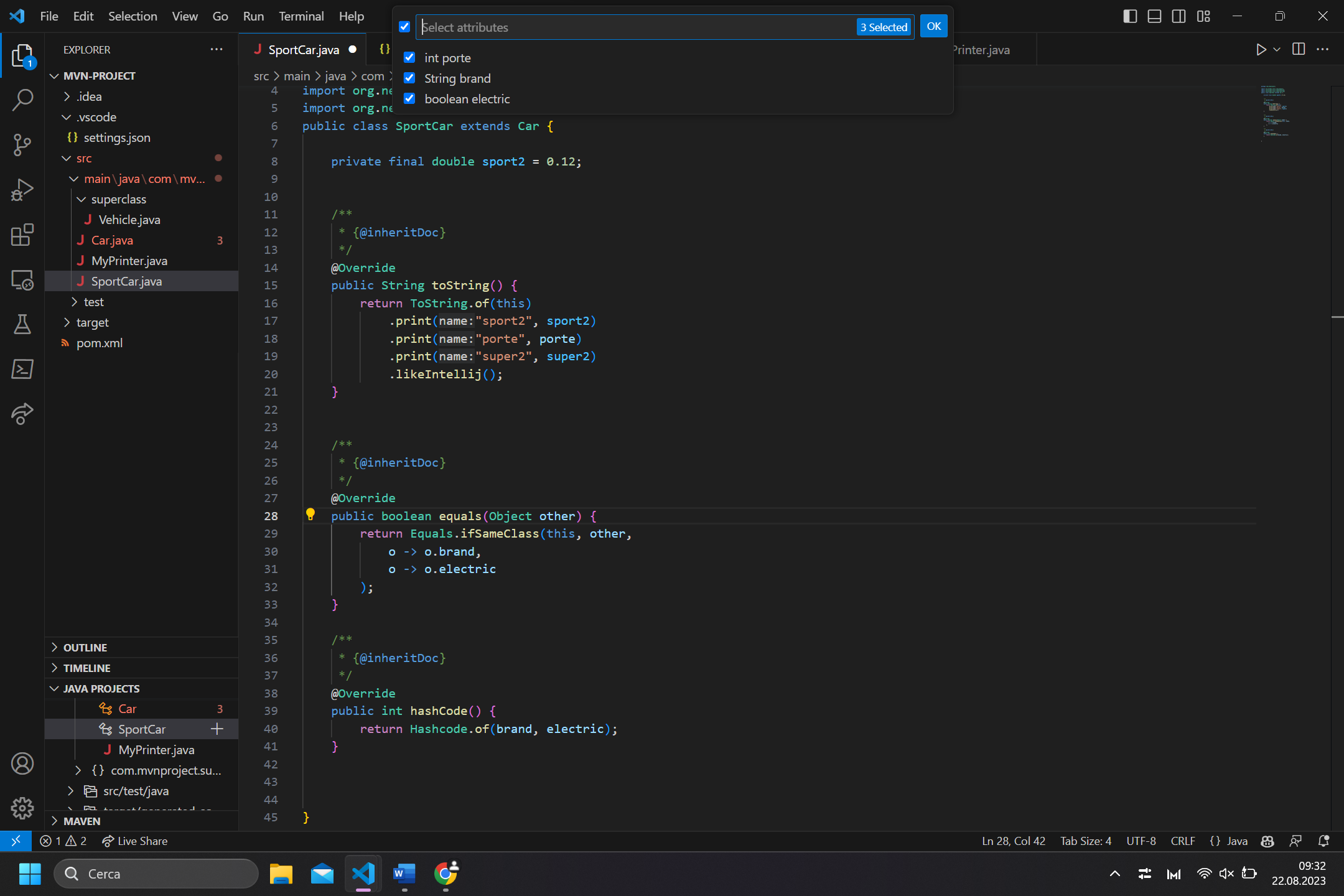


Figura 35 Generazione withField() - selezione attributi

#### Risultato

Il codice dei vari withField() viene generato ed inserito e indentato correttamente all’interno del file java in cui è stata richiesta la generazione.

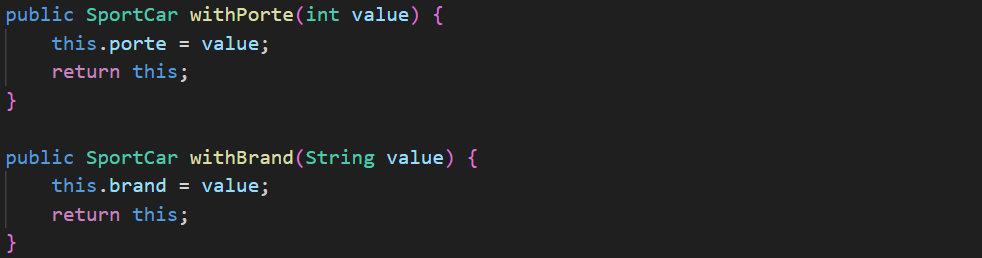


Figura 36 withField() - risultato

#### Generazione codice

Per la generazione dei metodi withField() vengono passati al metodo *generateWithField()* l’array che contiene gli attributi modificabili selezionati dall’utente nei passaggi precedenti e il nome della classe. Successivamente viene inserito tramite *editBuilder* all’interno del file il codice appena generato.

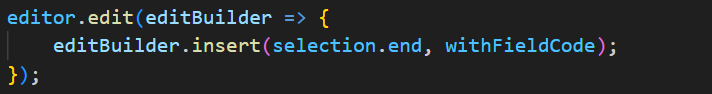


Figura 37 withField() - inserimento

#### Rigenerazione codice

Al contrario dei metodi *toString()*, *equals()* e *hashCode()*, i metodi *withField()* non hanno la possibilità di essere rigenerati. Questa scelta è stata fatta perché il loro contenuto non dovrebbe essere soggetto a modifiche come può invece accadere negli altri casi.

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.).