

SUPSI

Sviluppo di una estensione per VS Code

Studente/i

Beffa Bryan

Relatore

Coluzzi Massimo

Correlatore

-

Committente

Coluzzi Massimo

Corso di laurea

Ingegneria informatica

Codice progetto

C10652

Anno

2022/2023

Data

STUDENTSUPSI

Indice generale

1	Introduzione	6
2	Stato dell'arte	8
2.1	IDE	8
2.1.1	Tipologie di IDE	9
2.2	Visual Studio Code	9
2.2.1	Linguaggi di programmazione	9
2.2.2	Estensibilità	9
2.2.3	Strumenti di debugging	10
2.2.4	Gestione dei progetti	10
2.3	Extension Pack for Java	10
2.3.1	Language Support for Java	10
3	Motivazione e contesto	12
3.1	Programmazione fluent	12
3.2	Nerd4J	12
4	Problema	13
4.1	Obiettivo e requisiti	13
4.1.1	Conoscere ed utilizzare VS Code	13
4.1.2	Comprensione sviluppo estensioni	13
4.1.3	Studio Extension Pack for Java	13
4.1.4	Generazione automatica di codice	13
4.1.5	Creazione e sviluppo estensione	14
5	Implementazione e sviluppo	15
5.1	FileAnalyzer	15
5.1.1	Compatibilità	16
5.1.2	Schema generale	17
5.1.3	Costruttore	18
5.1.4	init()	18
5.1.5	isVisibleField()	18
5.1.6	getClassFields()	19
5.1.7	getParentsVisibleFields()	20
5.1.8	getVisibleFields()	20
5.1.9	main()	21
5.2	Creazione estensione	21
5.2.1	Installazione e configurazione Yeoman	22
5.3	Struttura estensione	23
5.3.1	Directory snippets	23

5.3.2	Directory src.....	24
5.4	Package.json.....	24
5.4.1	Supporto linguaggi	24
5.4.2	Definizione snippets	25
5.4.3	Definizione comandi.....	25
5.5	extension.ts.....	26
5.5.1	Comandi.....	26
5.5.2	getFields()	27
5.6	config.ts.....	28
5.7	jdkManagement.ts.....	29
5.7.1	getCurrentJDK().....	29
5.7.2	setWorkspaceJDK().....	29
5.8	codeGenerator.ts.....	30
5.8.1	checkIfMethodAlreadyExists()	31
5.8.2	getPackageName()	32
5.8.3	replaceOldCode()	32
5.8.4	checkJavadocComment().....	33
5.8.5	generateToStringCode()	34
5.8.6	generateEquals()	34
5.8.7	generateHashCode().....	34
5.8.8	generateWithFields().....	35
5.8.9	getIndentation()	35
5.8.10	insertTab()	36
5.9	Generazione codice	36
5.9.1	toString()	37
5.9.2	equals() e hashCode()	42
5.9.3	withField()	44
6	Test e risultati.....	46
7	Conclusioni	47
7.1	Sviluppi futuri	47
8	Sitografia	48

Indice delle figure

Figura 1 - Esempio output – FileAnalyzer	16
Figura 2 - schema comunicazione VS Code - java.....	17
Figura 3 - Assegnazione campi.....	17
Figura 4 - getClassFields() - logica.....	19
Figura 5 - getParentsVisibleFields () - logica.....	20
Figura 6 - main() – logica	21
Figura 7 - Yeoman - creazione estensione.....	22
Figura 8 – struttura modificata.....	23
Figura 9 - directory snippets.....	23
Figura 10 - directory src	24
Figura 11 - definizione comportamento comandi.....	26
Figura 12 – subscriptions	27
Figura 13 - costruzione comando java	27
Figura 14 Comando generate.....	36
Figura 15 - Generazione toString() - selezione attributi.....	37
Figura 16 - Generazione toString() - scelta layout.....	37
Figura 17 toString() – risultato.....	38
Figura 18 toString() - inserimento codice	38
Figura 19 toString() - sostituzione codice.....	38
Figura 20 - Printer snippet - implementazione.....	39
Figura 21 - Printer snippet - package.json configuration	39
Figura 22 - Printer snippet - suggerimento	39
Figura 23 - Printer snippet - codice esempio.....	41
Figura 24 - Printer snippet - risultato esempio.....	41
Figura 25 Generazione equals() e hashCode() - selezione attributi	42
Figura 26 Opzione creazione metodo hashCode	42
Figura 27 equals() e hashCode() – risultato	42
Figura 28 equals() e hashCode() - inserimento codice.....	43
Figura 29 equals() e hashCode() - regenerate	43
Figura 30 equals() e hashCode() – sostituzione codice	43
Figura 31 Generazione withField() - selezione attributi	44
Figura 32 withField() - risultato	44
Figura 33 withField() - inserimento	44

Abstract

Negli ultimi tempi, la "programmazione fluent" è diventata sempre più popolare tra gli sviluppatori. Questo approccio consente di scrivere codice in modo più chiaro e semplice, grazie all'utilizzo di sequenze di istruzioni concatenate e facilmente leggibili. Nerd4J è un framework che utilizza l'approccio della programmazione fluent, offrendo agli sviluppatori la possibilità di costruire applicazioni ad alte prestazioni in Java attraverso l'uso di codice fluent.

Questo articolo descrive lo sviluppo e la creazione di un'estensione per Visual Studio Code (VSCode) che consenta la generazione automatica di codice Java, in particolare sfruttando le classi messe a disposizione dal framework Nerd4J. L'obiettivo è fornire agli sviluppatori uno strumento che semplifichi la creazione di codice ripetitivo e tedioso, consentendo loro di concentrarsi su aspetti più significativi dello sviluppo.

L'estensione è in grado di riconoscere il codice sorgente di una classe Java all'interno dell'ambiente di sviluppo VSCode. Una volta individuata la classe, l'estensione sfrutta la reflection di Java per identificare tutti i campi visibili, inclusi quelli ereditati dalla gerarchia di classi. Gli sviluppatori hanno la possibilità di selezionare un insieme dei campi rilevati e poter generare i metodi *toString()*, *equals()*, *hashCode()* e *withField()*.

L'estensione mira a semplificare il processo di sviluppo di applicazioni Java attraverso l'automazione di compiti ripetitivi e la generazione di codice standardizzato, consentendo agli sviluppatori di concentrarsi su aspetti più creativi e significativi della programmazione, come ad esempio la logica e l'algoritmica.

I risultati ottenuti sono stati positivi e sono stati raggiunti gli obiettivi stabiliti per il progetto, facilitando e semplificando la programmazione attraverso la generazione e sostituzione automatica di codice. Vi è la possibilità di utilizzare snippet di codice per velocizzare le operazioni di import delle classi e aggiunta delle dipendenze Nerd4J. È stata inoltre aggiunta una parte di gestione delle jdk per rendere l'estensione più dinamica e compatibile con le diverse versioni di java.

Infine, questo documento parla di un possibile sviluppo futuro dell'estensione, in particolare delle modifiche e aggiunte che potrebbero essere implementate in futuro.

Abstract (EN)

Lately, "fluent programming" has become increasingly popular among developers. This approach allows for writing code in a clearer and simpler manner, thanks to the use of concatenated and easily readable sequences of instructions. Nerd4J is a framework that employs the fluent programming approach, offering developers the opportunity to build high-performance Java applications through the use of fluent code.

This article describes the development and creation of an extension for Visual Studio Code (VSCode) that enables the automatic generation of Java code, specifically leveraging the classes provided by the Nerd4J framework. The goal is to provide developers with a tool that streamlines the creation of repetitive and tedious code, allowing them to focus on more meaningful aspects of development.

The extension is capable of recognizing the source code of a Java class within the VSCode development environment. Once the class is identified, the extension utilizes Java reflection to identify all visible fields, including those inherited from the class hierarchy. Developers have the option to select a set of detected fields and generate the `toString()`, `equals()`, `hashCode()`, and `withField()` methods.

The extension aims to simplify the Java application development process by automating repetitive tasks and generating standardized code, enabling developers to concentrate on more creative and significant programming aspects such as logic and algorithms.

The achieved results have been positive, and the project's established objectives have been met by facilitating and simplifying programming through automatic code generation and replacement. There is the possibility of using code snippets to expedite operations like importing classes and adding Nerd4J dependencies. Additionally, a part for JDK management has been included to make the extension more dynamic and compatible with different Java versions.

Finally, this document discusses potential future development of the extension, particularly the changes and additions that could be implemented down the line.

1 Introduzione

Questo progetto si focalizza sulla creazione di un'estensione dedicata a Visual Studio Code (VSCode) finalizzata alla generazione automatica di codice Java. L'obiettivo primario di questa estensione è semplificare in modo significativo il processo di sviluppo, consentendo ai programmatori di concentrarsi maggiormente sull'aspetto creativo della programmazione e di ridurre il carico di lavoro ripetitivo e noioso.

Nel contesto di questa iniziativa, l'estensione è stata ideata per svolgere compiti specifici basati sul codice sorgente di una classe Java fornita:

1. Individuare e classificare i campi: l'estensione è in grado di analizzare il codice e individuare tutti i campi visibili all'interno di una classe Java. Questo include non solo i campi propri della classe ma anche quelli ereditati dalla gerarchia di classi. Questa caratteristica è estremamente utile in quanto semplifica il processo di selezione dei campi per ulteriori manipolazioni.
2. Personalizzazione selettiva: l'estensione offre agli sviluppatori la flessibilità di selezionare un sottoinsieme specifico di campi dalla lista ricavata in precedenza. Questo è particolarmente utile quando si desidera generare codice solo per determinati campi anziché per l'intera classe.
3. Generazione di metodi avanzati: una delle funzionalità principali dell'estensione è la generazione automatica di metodi avanzati, come *toString()*, *hashCode()* ed *equals()*, basati sui campi selezionati. Questo rende la gestione delle operazioni standard di generazione di codice più efficiente e accurata.
4. Supporto per Fluent Programming: la programmazione fluent è una metodologia sempre più popolare tra gli sviluppatori. L'estensione include la generazione di metodi di tipo "withField" che seguono l'approccio fluent programming. Questi metodi permettono una manipolazione agevole e concatenata dei campi, migliorando la leggibilità e la chiarezza del codice.

Per realizzare questi obiettivi, è necessario studiare l'ecosistema di sviluppo di VSCode, comprese le estensioni, e di esplorare dettagli tecnici come il funzionamento del Language Server Protocol (LSP). Inoltre, il progetto richiede l'apprendimento delle tecniche di generazione automatica di codice, concentrandosi sulle esigenze del linguaggio Java.

In sintesi, questa iniziativa rappresenta un'occasione per acquisire conoscenze approfondite nell'ambito di Visual Studio Code, esplorare l'implementazione di estensioni avanzate e sviluppare abilità nel creare soluzioni che semplifichino in modo considerevole il processo di sviluppo di applicazioni Java.

2 Stato dell'arte

Questo capitolo introduce il contesto relativo agli ambienti di sviluppo integrato (IDE), fondamentali strumenti utilizzati dagli sviluppatori per scrivere, testare e debuggare il codice in modo efficiente. In particolare, questo capitolo esplora le caratteristiche chiave degli IDE, le loro tipologie e offre una panoramica dettagliata dell'editor di codice Visual Studio Code (VS Code).

Viene analizzata l'estensione "Extension pack for Java", una raccolta di estensioni ideata e realizzata direttamente dal team di sviluppo di Microsoft. Questo pacchetto di estensioni è stato creato con l'obiettivo di fornire un supporto completo e specializzato per il processo di sviluppo di applicazioni Java all'interno di Visual Studio Code.

Viene esplorato anche il concetto di Language Server Protocol (LSP) implementato dall'Extension pack for Java, analizzando i vantaggi ed il contesto in cui viene normalmente inserito.

2.1 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.

2.1.1 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: specifici 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.

2.2 Visual Studio Code

Visual Studio Code è un editor di codice sorgente gratuito, estensibile e multiplatforma 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.3 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 parti 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*

2.3.1 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.

2.3.2 Language Server Protocol

L'extension pack for Java implementa un Language Server Protocol che permette l'analisi del codice sorgente e altre funzionalità attraverso dei server Java. L'utilizzo del LSP rende i server indipendenti dall'IDE, permettendo agli sviluppatori di concentrarsi sullo sviluppo senza pensare all'integrazione con i diversi ambienti di sviluppo.

Gli LSP vengono solitamente integrati in progetti di grandi dimensioni e che richiedono frequenti modifiche, rendendo però il sistema, in termini di risorse, più pesante e meno veloce rispetto all'utilizzo dei file sorgenti senza LSP. L'uso di un LSP è più vantaggioso quando si desidera una maggiore automazione, analisi avanzata e un'esperienza di sviluppo più completa. Tuttavia, per progetti più semplici, lavorare direttamente con i file di codice sorgente può essere sufficiente e può evitare una complessità eccessiva nell'architettura del progetto.

3 Motivazione e contesto

Nel mondo della programmazione, la scrittura di codice non è solo un insieme di istruzioni che vengono eseguite da un computer, ma è anche un mezzo di comunicazione tra il programmatore e la macchina. La programmazione fluida è un concetto che va oltre la semplice sintassi corretta e la logica dell'algoritmo, ma si tratta di scrivere codice in modo chiaro, elegante e facile da leggere e mantenere. L'obiettivo è quello di sviluppare un'estensione di VSCode che faciliti la creazione e generazione di metodi fondamentali come *toString()*, *equals()*, *hashCode()* e *withField()* sfruttando il concetto di programmazione fluent implementato dal framework "Nerd4J" sviluppato per java .

3.1 Programmazione fluent

La programmazione fluida è uno stile di scrittura del codice che mira a rendere le istruzioni più leggibili e simili al linguaggio naturale. Si basa sull'uso di catene di chiamate di metodo, consentendo la concatenazione di operazioni in sequenza. Questo stile è particolarmente utile quando si lavora con operazioni complesse o nidificate, poiché migliora la chiarezza e la comprensibilità del codice .La programmazione fluida cerca di creare un flusso di istruzioni che sembrano frasi strutturate, migliorando la leggibilità e facilitando la manutenzione del codice.

3.2 Nerd4J

Nerd4J rappresenta un progetto che si pone l'obiettivo di semplificare il processo di sviluppo software attraverso l'offerta di un insieme di librerie e framework appositamente creati. L'idea alla base di Nerd4J è affrontare il problema del codice ripetitivo, che spesso rappresenta uno dei principali ostacoli nello sviluppo efficiente e di alta qualità. Questo progetto mira a eliminare il compito di dover continuamente riscrivere il medesimo codice di base, fornendo soluzioni predefinite per le operazioni più comuni e astrazioni per affrontare problemi ricorrenti. In tal modo, gli sviluppatori possono concentrarsi su aspetti più creativi e complessi del loro lavoro, lasciando alle librerie Nerd4J la gestione delle parti di codice che possono essere automatizzate. Nerd4J punta a migliorare la produttività, la manutenibilità e la leggibilità del codice, facilitando il processo di sviluppo e consentendo agli sviluppatori di dedicare più tempo allo sviluppo della logica.

4 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 sui 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.

4.1 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.

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.1.5 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

5 Implementazione e sviluppo

In questo capitolo viene trattata la parte di implementazione e sviluppo dell'estensione. Vengono motivate le scelte e le decisioni prese sull'implementazione con lo scopo di spiegare come è stata svolta la parte di lavoro pratico. Viene spiegata passo per passo la parte di creazione di un'estensione per VSCode dall'inizio.

È possibile consultare le parti di architettura del sistema, analisi del codice java, schema di funzionamento, struttura del progetto e relativi file sorgenti e la spiegazione del funzionamento dei metodi di generazione del codice.

5.1 Architettura del sistema

Non essendo un progetto di grandi dimensioni ed estremamente complesso è stato deciso di implementare le funzionalità di analisi e modifica del codice senza l'utilizzo di un LSP. Questa decisione è stata presa considerando i seguenti fattori:

- Dimensione: le dimensioni del progetto sono ridotte ed è sconsigliato utilizzare un LSP in quanto rende tutto il sistema più appesantito.
- Complessità architettura: sempre per la dimensione del progetto è stata presa la decisione di non rendere l'architettura troppo complessa.
- Risorse: la comunicazione degli LSP rende il sistema più pesante e lento in termini di risorse è stato preferito un approccio più semplice e leggero, ovvero l'utilizzo dei file sorgenti implementati direttamente su VSCode.
- Supporto multiplatforma: in questo progetto il supporto multiplatforma non è stato considerato in quanto l'estensione è richiesta specificatamente per Visual Studio Code.
- Complessità implementazione: implementare un LSP è inoltre più complesso e, se non necessario, è sconsigliato l'utilizzo in progetti in cui non è considerato necessario.

5.2 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 cartella 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

```
Car  
String brand  
boolean elettrica  
String super2
```

Figura 1 - Esempio output – FileAnalyzer

5.2.1 Compatibilità

La classe FileAnalyzer è stata compilata 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.

5.2.2 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 comando illustrato nello schema tramite la libreria *child_process*, e attende che venga restituito il risultato sullo standard output.

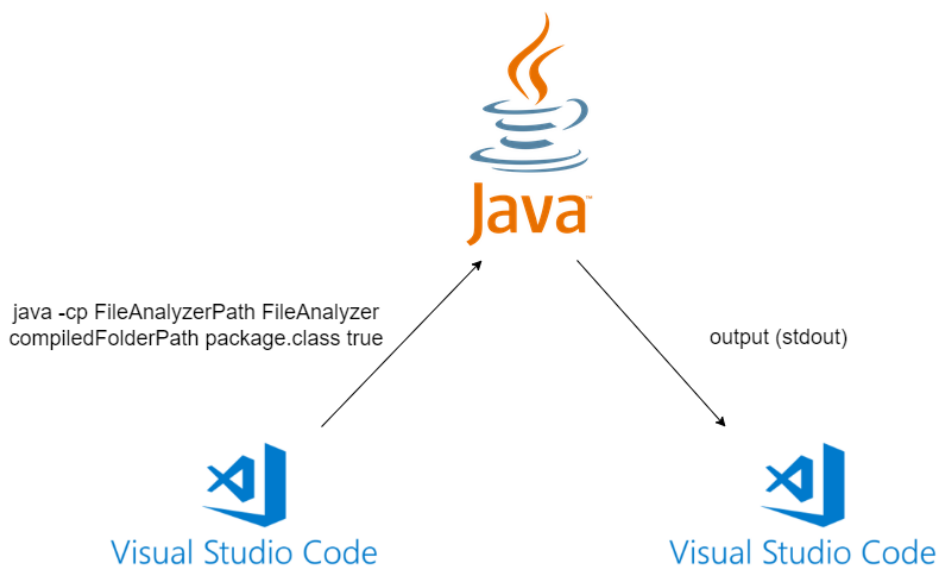


Figura 2 - schema comunicazione VS Code - java

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.

```
//remove all options
options = [];
className = outputList[0].trim();
for (let i = 1; i < outputList.length; i++)
    options.push({ label: outputList[i].trim(), picked: true });
```

Figura 3 - Assegnazione campi

5.2.3 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:

- **compiledFilePath:** 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(compiledFilePath, className, editableFields);
```

5.2.4 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:

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

5.2.5 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

5.2.6 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:

```
for (Field field : declaredFields) {  
    if (!Modifier.isStatic(field.getModifiers())) {  
  
        // check if the field is final  
        if (editableFields) {  
            if (!Modifier.isFinal(field.getModifiers()))  
                fields.add(field.getType().getSimpleName() + " " + field.getName());  
        } else  
            fields.add(field.getType().getSimpleName() + " " + field.getName());  
    }  
}
```

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();
```

5.2.7 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:

```
// get all the public fields of the parent classes
while (!clazz.getSuperclass().equals(obj:Object.class)) {

    // get all the fields of the parent classes
    clazz = clazz.getSuperclass();

    Field[] fields = clazz.getDeclaredFields();
    for (Field field : fields) {

        if (isVisibleField(field) && !Modifier.isPrivate(field.getModifiers())) {

            // check if the field is final
            if (editableFields) {
                if (!Modifier.isFinal(field.getModifiers()))
                    parentsClassPublicFields.add(field.getType().getSimpleName() + " " + field.getName());
            } else
                parentsClassPublicFields.add(field.getType().getSimpleName() + " " + field.getName());
        }
    }
}
```

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();
```

5.2.8 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();
```

5.2.9 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.

```
// create the file analyzer
FileAnalyzer fileAnalyzer = new FileAnalyzer(compiledFilePath, className, editableFields);

// get all visible fields
List<String> visibleFields = fileAnalyzer.getVisibleFields();
for (String field : visibleFields)
    System.out.println(field);
```

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()*.

5.3 Creazione estensione

Per la creazione di un'estensione in VS Code è stato scelto l'utilizzo di *Yeomen*, 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 comando, 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.

5.3.1 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)*.

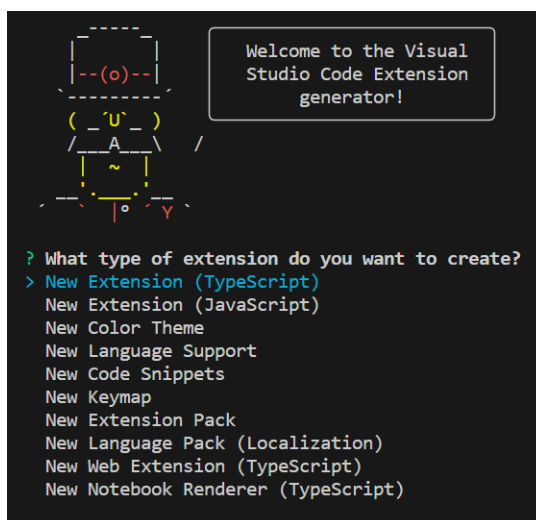


Figura 7 - Yeoman - creazione estensione

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

5.4 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:

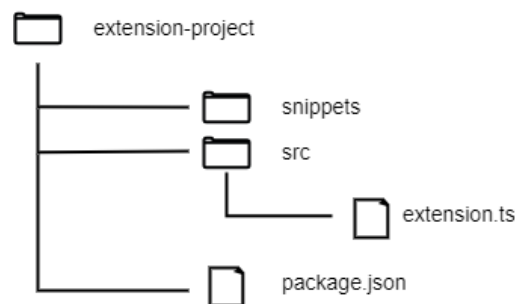


Figura 8 – struttura modificata

5.4.1 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 *dependencies* 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.

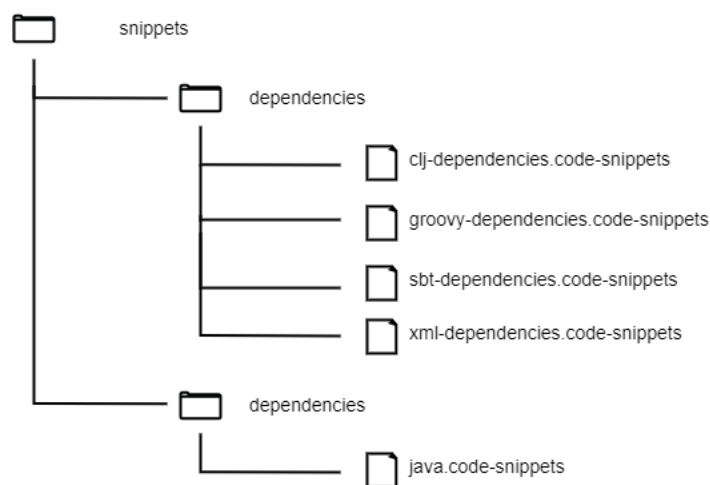


Figura 9 - directory snippets

5.4.2 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* è invece presente una gerarchia che si occupa di eseguire i test dell'estensione.

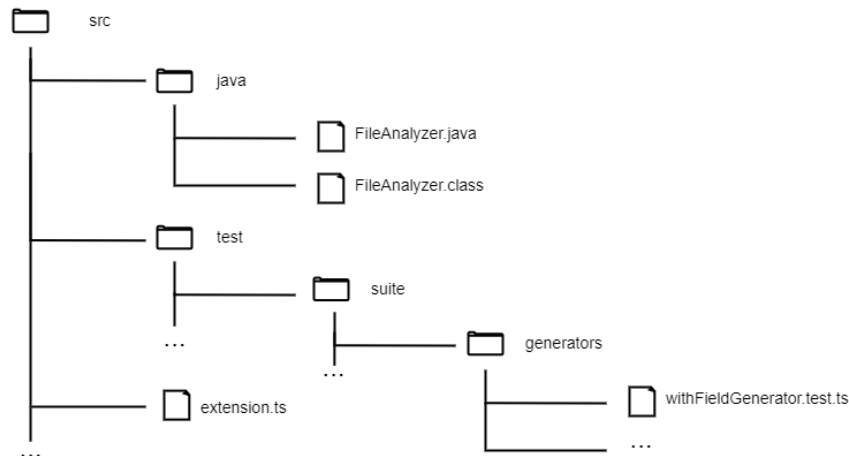


Figura 10 - directory src

5.5 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.

5.5.1 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"
    ]
  }
]

```


5.5.2 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"  
  },  
  ...  
]
```

5.5.3 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"  
  }  
]
```

5.6 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.

5.6.1 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.

```
//generate toString command
const toString = vscode.commands.registerCommand('nerd4j-extension.generateToString', async () => { ...
});

//generate with field command
const withField = vscode.commands.registerCommand('nerd4j-extension.generateWithField', async () => { ...
});
```

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 contenente 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.

```
context.subscriptions.push(setJDKWorkspace);
context.subscriptions.push(checkCurrentJDK);
context.subscriptions.push(recompileFileAnalyzer);
context.subscriptions.push(toString);
context.subscriptions.push(withField);
context.subscriptions.push(allMethods);
context.subscriptions.push(equals);
context.subscriptions.push(setCustomCompiledFolder);
context.subscriptions.push(deleteCustomCompiledFolder);
context.subscriptions.push(showContextMenu);
```

Figura 12 – subscriptions

5.6.2 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.

```
// get package name
const packageName = getPackageName(activeEditor.document.getText());
const classDefinition = (packageName) ? `${packageName}.${fileName.split('.')[0]}` : fileName.split('.')[0];
const javaCommand = `${JAVA_COMMAND} ${fullCompiledPath} ${classDefinition} ${editableField}`;
```

Figura 13 - costruzione comando java

Valore di ritorno:

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

5.7 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()'

5.8 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()*.

```
export const jdkQuickFix = { title: 'Set workspace JDK', command: 'nerd4j-extension.setWorkspaceJDK' };
```

Figura 14 jdkManagement – jdkQuickFix

5.8.1 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();
```

5.8.2 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);
```

5.9 path.ts

Il file in *path.ts* svolge una serie di funzioni legate alla gestione dei percorsi della cartella di output di compilazione. Questo file mette a disposizione metodi che servono a determinare la posizione della cartella contenente le classi java compilate (.class).

Sono definite delle costanti che indicano i percorsi di default utilizzati dai framework più comuni:

- `./target/classes`
- `./build/classes`

È presente, inoltre, una variabile per configurare un percorso personalizzato qualora fosse richiesto esplicitamente dall'utente attraverso il comando "*Nerd4J: set custom compiled files folder*". È possibile utilizzare nuovamente il percorso di default eseguendo invece il comando "*Nerd4J: delete custom compiled files folder*".

5.9.1 setCustomizedPath()

Descrizione:

Questo metodo permette di definire il percorso di una cartella personalizzata in cui verranno inseriti i file java compilati.

Parametri richiesti:

- **path**: percorso della cartella personalizzata

Esempio di utilizzo:

```
setCustomizedPath(compiledFolder);
```

5.9.2 deleteCustomizedPath()

Descrizione:

Questo metodo permette di eliminare il percorso personalizzato di una cartella utilizzando le cartelle di default.

Esempio di utilizzo:

```
deleteCustomizedPath();
```

5.10 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)`

5.10.1 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)) {  
    ...  
}
```

5.10.2 getPackageName()

Descrizione:

Metodo che ritorna, 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());
```

5.10.3 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.

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

const range = new vscode.Range(editor!.document.positionAt(oldCodeIndex), oldCodeLastIndex);
await editor?.edit(editBuilder => {
    editBuilder.replace(range, newCode);
});
```

Figura 15 replaceOldCode() - logica

Esempio di codice:

```
await replaceOldCode(toStringRegExp, toStringCode);
```


5.10.4 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 "{".

```
for (index; index >= 0; index--) {
    const currentChar = editorText.charAt(index);

    if (currentChar === '/') {
        if (editorText.charAt(index - 1) === '*') {
            ignoreChar = true;
        }
    }

    if (!ignoreChar && (currentChar === ';' || currentChar === '}' || currentChar === '{')) {
        return oldCodeIndex;
    }

    if (currentChar === '*') {
        if (editorText.charAt(index - 1) === '*') {
            if (editorText.charAt(index - 2) === '/') {
                return index - 2;
            }
        }
    }
}
```

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);
```

5.10.5 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);
```

5.10.6 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 *equals()*

Valore di ritorno:

Stringa contenente il metodo *equals()*.

Esempio di utilizzo:

```
const equalsCode = await generateEquals(selectedAttributes);
```

5.10.7 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);
```

5.10.8 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);
```

5.10.9 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.

```
for (let i = 0; i < textUntilCursor?.length; i++) {  
  if (textUntilCursor[i] === '{') {  
    indentation++;  
  } else if (textUntilCursor[i] === '}') {  
    indentation--;  
  }  
}
```

Figura 17 getIndentation() - logica

Esempio di utilizzo:

```
const tabs = insertTab(getIndentation());
```

5.10.10 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());
```

5.11 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.

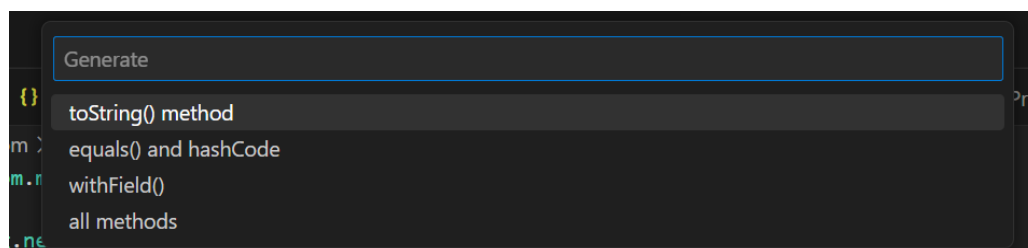


Figura 18 Comando generate

5.11.1 toString()

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

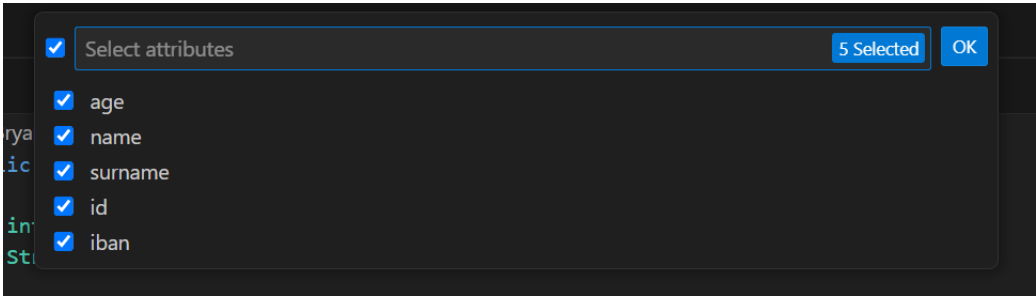


Figura 19 - Generazione toString() - selezione attributi

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.

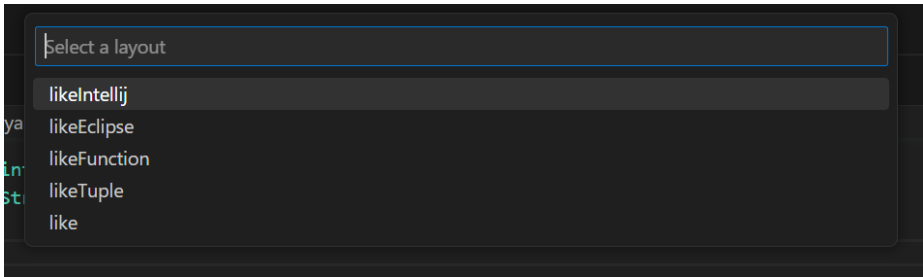


Figura 20 - Generazione toString() - scelta layout

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.

```
/**
 * {@inheritDoc}
 */
@Override
public String toString() {
    return ToString.of(this)
        .print(name:"sport2", sport2)
        .print(name:"porte", porte)
        .print(name:"super2", super2)
        .likeIntelliJ();
}
```

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.

```
await editor.edit(editBuilder => {

    // add import if is not present
    if (!checkIfImportExists(TO_STRING_IMPORT)) {
        editBuilder.insert(new vscode.Position(1, 0), `\n${TO_STRING_IMPORT}`);
    }
    editBuilder.insert(selection.end, toStringCode);
});
```

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 vecchio codice con quello appena generato.

```
if (checkIfMethodAlreadyExists(TO_STRING_SIGNATURE)) {
    const ans = await vscode.window.showInformationMessage("The toString() method is already implemented.", "Regenerate", "Cancel");
    if (ans === "Regenerate") {
        await replaceOldCode(toStringRegExp, toStringCode);
        vscode.window.showInformationMessage("toString() method regenerated");
    }
    return;
}
```

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 estensione `.code-snippets` che permette la creazione di parti di codice che possono essere espansive all'interno dell'editor durante l'implementazione del codice.

```
{
  "Printer": {
    "prefix": [
      "Printer",
      "ToString.Printer"
    ],
    "body": [
      "new ToString.Printer() {",
      "\n\t@Override",
      "...",
    ],
    "description": "Example of an implementation of a customizable layout using ToString.Printer",
  }
}
```

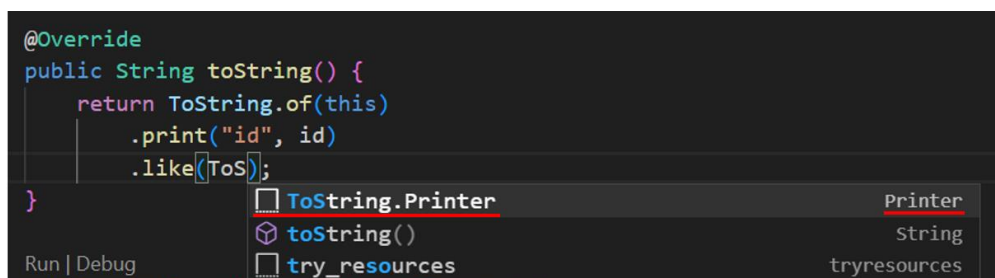
Figura 24 - Printer snippet - implementazione

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

```
"contributes": {
  "snippets": [
    {
      "language": "java",
      "path": "./snippets/java.code-snippets"
    }
  ],
}
```

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.



```
@Override
public String toString() {
    return ToString.of(this)
        .print("id", id)
        .like(ToString);
}
```

The suggestion list shows:

- `ToString.Printer` (Printer)
- `toString()` (String)
- `try_resources` (tryresources)

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
- firstDelimiter: carattere o insieme di caratteri che indica l'inizio della stampa dell'attributo/degli attributi
- lastDelimiter: 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 variabili, 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:

```
@Override
public String toString() {
    return ToString.of(this)
        .print("name", name)
        .print("id", id)
        .like(new ToString.Printer() {

            @Override
            public String apply(ToString.Configuration configuration) {

                String separator = ";";
                String firstDelimiter = "(";
                String lastDelimiter = ")";
                String equalityOperator = "=>";
                StringBuilder sb = new StringBuilder();

                // get the class name
                if (configuration.customClassName() != null)
                    sb.append(configuration.customClassName());
                else if (configuration.fullClassPath())
                    sb.append(configuration.target().getClass().getCanonicalName());
                else
                    sb.append(configuration.target().getClass().getSimpleName());
                sb.append(firstDelimiter);
                Iterator<ToString.Configuration.Field> fields = configuration.fields().iterator();

                while (fields.hasNext()) {

                    ToString.Configuration.Field field = fields.next();
                    sb.append(field.name).append(equalityOperator).append(field.value);

                    // Add separator if there are more fields
                    if (fields.hasNext())
                        sb.append(separator).append(str:" ");
                }
                sb.append(lastDelimiter);
                return sb.toString();
            }

        });
}
```

Figura 27 - Printer snippet - codice esempio

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

```
Car(name=>Prova; id=>0)

Process finished with exit code 0
```

Figura 28 - Printer snippet - risultato esempio

5.11.2 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()`.

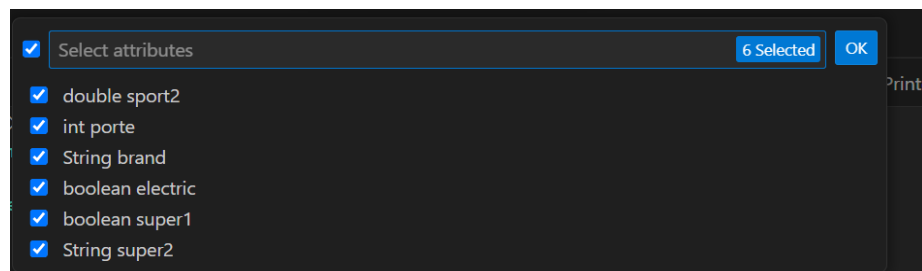


Figura 29 Generazione `equals()` e `hashCode()` - selezione attributi

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.

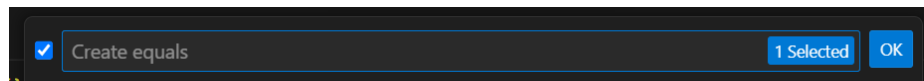


Figura 30 Opzione creazione metodo `hashCode`

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.

```
/**
 * {@inheritDoc}
 */
@Override
public boolean equals(Object other) {
    return Equals.ifSameClass(this, other,
        o -> o.brand,
        o -> o.electric
    );
}

/**
 * {@inheritDoc}
 */
@Override
public int hashCode() {
    return Hashcode.of(brand, electric);
}
```

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:

```
await editor.edit(editBuilder => {
    // add imports if is not present
    if (!checkIfImportExists(EQUALS_IMPORT)) {
        editBuilder.insert(new vscode.Position(1, 0), `\n${EQUALS_IMPORT}`);
    }
    if (!checkIfImportExists(HASHCODE_IMPORT) && createHashCode) {
        editBuilder.insert(new vscode.Position(1, 0), `\n${HASHCODE_IMPORT}`);
    }

    editBuilder.insert(selection.end, code);
});
```

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.

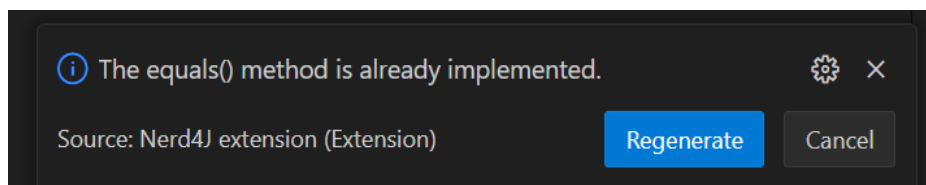


Figura 33 equals() e hashCode() - regenerate

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.

```
// delete old code
if (regenerateEquals) {
    await replaceOldCode(equalsRegExp, equalsCode);
    vscode.window.showInformationMessage("equals() method regenerated");
}
if (regenerateHashCode) {
    await replaceOldCode(hashCodeRegExp, hashCode);
    vscode.window.showInformationMessage("hashCode() method regenerated");
}
```

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

5.11.3 withField()

Per la generazione dei metodi `withField` è necessario eseguire nuovamente il comando “*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.

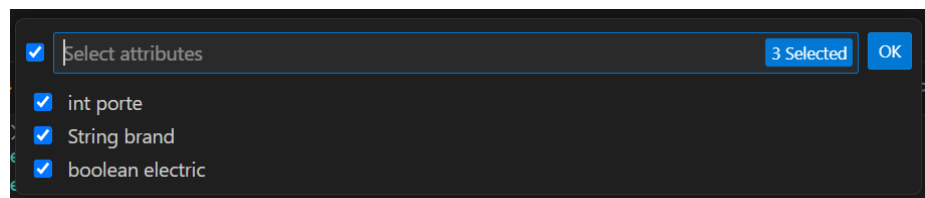


Figura 35 Generazione `withField()` - selezione attributi

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.

```
public SportCar withPorte(int value) {
    this.porte = value;
    return this;
}

public SportCar withBrand(String value) {
    this.brand = value;
    return this;
}
```

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.

```
editor.edit(editBuilder => {
    editBuilder.insert(selection.end, withFieldCode);
});
```

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.

6 Test e risultati

I test presentati sono stati sviluppati utilizzando il framework di testing 'node:test', che fornisce un ambiente per eseguire test unitari e di integrazione in progetti Node.js. Questo framework offre strumenti per definire e organizzare i test in modo chiaro e strutturato.

```
describe('generateEquals', () => {
  const tabs = ' ';

  // test equals with selected attributes
  it('should generate the correct code for equals method', async () => { ...
  });

  // test equals with empty selected attributes
  it('should return an empty string when selected attributes are empty', async () => { ...
  });
});
```

Figura 38 Esempio test - generateEquals

La funzione *describe()* è utilizzata per raggruppare e descrivere un insieme di test correlati. Fornisce un modo per organizzare gerarchicamente i test e per chiarire il contesto in cui vengono eseguiti. All'interno di una chiamata *describe()*, vengono dichiarati uno o più test utilizzando la funzione *it()*.

La funzione *it()* è utilizzata per definire un singolo test unitario. Ogni chiamata *it()* rappresenta un singolo caso di test e contiene l'implementazione delle asserzioni che verificano il comportamento del codice in esame.

I test sono stati sviluppati per i metodi *generateToString()*, *generateEquals()*, *generateHashCode()*, *packageName()* e *generateWithFiel()*. Per ogni test è stato confrontato l'output previsto con l'output prodotto per verificare che i metodi abbiano il comportamento atteso, in base alla selezione degli attributi.

```
// test equals with selected attributes
it('should generate the correct code for equals method', async () => {
  const selectedAttributes = ['String name', 'int age', 'boolean isActive'];
  let expectedCode = `\\n${tabs}/**\\n${tabs} * `
    + `@inheritDoc\\n${tabs} */\\n${tabs}@Override\\n${tabs}public boolean equals(Object other) `
    + `{\\n${tabs}\\treturn Equals.ifSameClass(this, other,`
    + `\\n${tabs}\\t\\to -> o.name, `
    + `\\n${tabs}\\t\\to -> o.age, `
    + `\\n${tabs}\\t\\to -> o.isActive`
    + `\\n${tabs}\\t);\\n${tabs}}\\n`;

  const generatedCode = await generateEquals(selectedAttributes);
  assert.strictEqual(`${generatedCode}`.trim(), expectedCode.trim());
});

// test equals with empty selected attributes
it('should return an empty string when selected attributes are empty', async () => {
  const selectedAttributes: string[] = [];
  const generatedCode = await generateEquals(selectedAttributes);

  let expectedCode = `\\n${tabs}/**\\n${tabs} * `
    + `@inheritDoc\\n${tabs} */\\n${tabs}@Override\\n${tabs}public boolean equals(Object other) `
    + `{\\n${tabs}\\treturn Equals.ifSameClass(this, other);`
    + `expectedCode += `);\\n${tabs}}\\n`;

  assert.strictEqual(generatedCode, expectedCode);
});
```

Figura 39 Esempio test

7 Conclusioni

Il progetto è stato sviluppato con l'obiettivo di migliorare l'efficienza e semplificare lo sviluppo di codice offrendo agli utenti un'estensione in grado di generare metodi *toString()*, *equals()*, *hashCode()* e *withField()* personalizzati sfruttando il concetto di programmazione fluent integrato all'interno delle librerie messe a disposizione da "Nerd4J".

Grazie all'implementazione dell'estensione di VSCode gli utenti hanno la possibilità di risparmiare tempo generando automaticamente questi metodi per concentrarsi maggiormente sulla logica del codice.

I risultati ottenuti sono positivi e sono stati raggiunti gli obiettivi stabiliti per il progetto, facilitando e semplificando la programmazione attraverso la generazione e sostituzione automatica di codice. Il codice viene correttamente generato, indentato e, in caso di rigenerazione, sostituisce automaticamente i metodi già presenti, aggiungendo, se necessario, gli import richiesti.

Sono stati aggiunti degli snippet di codice che velocizzano le operazioni di aggiunta delle dipendenze Nerd4J per gli strumenti di gestione dei pacchetti *Apache Maven*, *Apache Buildr*, *Apache Ant*, *Groovy Grape*, *Grails*, *Leiningen* e *SBT*. È stata inoltre aggiunta una parte di gestione del Java Development Kit (JDK) per rendere l'estensione compatibile con le diverse possibili versioni di java.

7.1 Sviluppi futuri

Per quanto riguarda gli sviluppi futuri di questo progetto sarebbe molto interessante implementare nuovi snippet di codice relativi al framework "Nerd4J" velocizzando ulteriormente lo sviluppo.

In futuro potrebbe essere implementato un Language Server Protocol (LSP) che offre una struttura standardizzata per la comunicazione tra un ambiente di sviluppo, ad esempio Visual Studio Code, e i server di linguaggio. I server di occupano dell'analisi e l'elaborazione del codice sorgente in un linguaggio specifico. LSP viene utilizzato principalmente per progetti di grandi dimensioni e per l'utilizzo delle funzionalità messe a disposizione su più IDE. In questo progetto ho deciso di non utilizzare un Language Server Protocol per mantenere l'ambiente più leggero e veloce in termini di risorse. Ciò non esclude però che in futuro possa venir implementato e sostituire l'utilizzo dei file di codice sorgente senza LSP permettendo la creazione di estensioni anche per altri ambienti di sviluppo.

Al momento è consigliato utilizzare l'estensione di VSCode sviluppata per Nerd4J con il supporto dell' "Extension pack for java", in quanto quest'ultimo ad ogni salvataggio avvenuto in un file java crea il `.class` all'interno della cartella configurata per i file configurati. Utilizzando la reflection è quindi fondamentale essere in possesso dei file compilati. Sarebbe quindi interessante aggiungere una parte di gestione e creazione dei file compilati, al fine di non richiedere ulteriormente il supporto di librerie esterne. È comunque possibile utilizzare l'estensione senza la collaborazione dell' "Extension pack for java" richiedendo però la compilazione manualmente le classi java.

8 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

https://en.wikipedia.org/wiki/Fluent_interface, *Fluent interface*, 24.08.2023