ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Quarkdown

Typesetting versatile di documenti articolati

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Mirko Viroli

Candidato
Giorgio Garofalo

Correlatore
Prof. Gianluca Aguzzi

II Sessione di Laurea
Anno Accademico 2023-2024

Sommario

Il mercato dei linguaggi di markup e di sistemi di *typesetting*, in cui LaTeX è uno dei prodotti più usati, presenta molteplici strumenti che permettono la realizzazione ed esportazione di documenti a partire da un codice sorgente, tuttavia seguendo spesso sintassi *proprietarie* anziché adottandone una standard e familiare agli utenti.

Tra le sintassi 'standard di fatto' si colloca Markdown, estremamente diffuso nel mondo dell'open source. Sul suo ampio utilizzo si fonda il principio di Quarkdown, che vuole permettere la creazione di documenti complessi e versatili usando una sintassi popolare, ben nota e con una specifica formale al fine di appiattire la curva di apprendimento, ma allo stesso tempo garantendo anche un controllo d'insieme sul documento, che non è offerto dallo standard di Markdown.
Quarkdown diventa così un *dialetto* che introduce la possibilità di definire e chiamare **funzioni** all'interno di codice Markdown, rendendo così infinite le possibilità di personalizzazione di un documento che, su scelta dell'utente, avrà le caratteristiche estetiche, strutturali e funzionali di una presentazione, un articolo o di un libro.

La tesi qui presente affronterà la creazione del progetto, dalle motivazioni alla compilazione di un documento, dalla lettura del codice sorgente alla produzione dei file di output, attraverso una pipeline che verrà estensivamente approfondita.

Parole chiave: Markdown, markup, typesetting, Turing complete, generazione di documenti, presentazioni, libri.

Indice

Sommario **iii**

1 Introduzione **1**

2 Background **3**
2.1 Analisi dei competitor 3
2.1.1 Conclusioni 7

3 Analisi e pianificazione **9**
3.1 Sintassi 9
3.1.1 Chiamate a funzioni 9
3.2 Esempio pratico 13

4 Tipi di elementi **15**
4.1 Blocchi 15
4.2 Inline 19

5 Design **21**
5.1 Pipeline 21
5.2 Flavor 24

6 Implementazione (preambolo) **27**
6.1 Struttura 27
6.2 Test automatici 28

7 Lexing **29**
7.1 Design 30
7.1.1 Strategia di lexing 30
7.1.2 Architettura finale 33
7.2 Implementazione 34
7.2.1 Contenuto innestato 36

8 Parsing **37**
8.1 Design . . . 37
8.1.1 Visitor pattern . . . 40
8.1.2 Token visitor . . . 41
8.1.3 Architettura finale . . . 43
8.2 Implementazione . . . 44
8.2.1 Block token parser . . . 44
8.2.2 Inline token parser . . . 52

9 Scripting **57**
9.1 Design . . . 57
9.1.1 Espressioni e valori . . . 60
9.1.2 Chiamate . . . 63
9.1.3 Definizione . . . 64
9.1.4 Librerie e registrazione . . . 64
9.1.5 Espansione . . . 67
9.1.6 Gestione degli errori . . . 68
9.2 Implementazione . . . 70
9.2.1 Funzioni di flusso . . . 71

10 Attraversamento dell'albero **77**
10.1 Design . . . 77
10.1.1 Contesto e attributi . . . 77
10.1.2 Aggiornamento degli attributi . . . 78
10.2 Implementazione . . . 80

11 Rendering **83**
11.1 Design . . . 83
11.1.1 Tag builder . . . 85
11.1.2 Flavor . . . 88
11.1.3 Architettura finale . . . 89
11.2 Implementazione . . . 90

12 Post rendering **95**
12.1 Design . . . 95
12.1.1 Render wrapper . . . 95
12.1.2 Tipo di documento . . . 96
12.1.3 Dettagli del documento . . . 98
12.1.4 Risorse . . . 98
12.2 Implementazione . . . 99

13 Assemblaggio della pipeline **101**
13.1 Design 101
13.1.1 Esportazione su file 102
13.2 Implementazione 104

14 Dimostrazione **107**
14.1 Strumenti a confronto 107
14.1.1 Quarkdown vs. LaTeX 107
14.1.2 Quarkdown vs. AsciiDoc 110
14.1.3 Quarkdown vs. Quarto 112
14.1.4 Quarkdown vs. Typst 113
14.2 Presentazione di esempio 116

15 Conclusioni **127**
15.1 Piani futuri 129

131

Bibliografia **131**

Capitolo 1

Introduzione

Quarkdown è un linguaggio di markup che si pone come un'estensione di Markdown, linguaggio nato nel 2004 e reso popolare particolarmente dall'adozione nel mondo della documentazione open source. Regolato più tardi dalla specifica CommonMark, oggi è uno standard di fatto.

L'idea nasce dalla volontà di adattare una sintassi standard, semplice, leggibile e compatta come quella di Markdown ad un contesto più esteso e complesso, ossia quello dei documenti. Qui assumono ruoli rilevanti paginazione, estetica e funzionalità dinamiche, aspetti che Markdown non riesce autonomamente a soddisfare, in quanto linguaggio che definisce puramente la struttura del documento (paragonabile al ruolo che HTML assume in una pagina web).

Quarkdown estende la specifica introducendo nuovi elementi che arricchiscono l'esperienza dell'utente con nuove sintassi e funzionalità non previste dallo standard. Tra questi spiccano le **funzioni** che vengono eseguite a tempo di compilazione e mutano proprietà e struttura del documento, introducendo una componente dinamica che mira a rendere la scrittura del documento leggera, completa e priva di ripetizioni. Le funzioni aprono infinite porte alla possibilità di personalizzazione del documento, rendendo Quarkdown un trade-off bilanciato tra la semplicità di Markdown e la completezza di LaTeX: sono ad esempio disponibili, tra le tante, funzioni di layout, di I/O, matematiche e perfino di scripting.

Questa tesi tratterà in maniera dettagliata il percorso completo di un'istanza di esecuzione di Quarkdown su un generico file, dalla lettura alla produzione di un artefatto (un documento di tipo presentazione, libro o articolo), rappresentata da una *pipeline*, che verrà trattata nel dettaglio in sezione 5.1, composta da diversi piccoli stadi, ognuno dei quali affrontato a tutto tondo in un proprio capitolo di questo documento e diviso nelle due macrosezioni *Design* e *Implementazione*.

La tesi qui presente non vuole risultare un'effimera dimostrazione del comportamento del programma e della sua architettura interna, ma vuole affrontare gradualmente le fasi dello sviluppo, le difficoltà incontrate e le soluzioni adottate: non sarà raro trovare snippet di codice che verranno deprecati nella pagina immediatamente successiva a fronte di una decisione opportunamente documentata, al fine di far trasparire il processo costante di design e di sviluppo che è stato svolto e che è tuttora in evoluzione.

Nella parte finale, in sezione 14.1, verranno infine mostrati esempi pratici del linguaggio, al fine di mostrare la concreta utilizzabilità e semplicità del prodotto finale, adatto a una grande varietà di utenti.

Il programma, scritto in Kotlin ed eseguito sulla JVM, è open source e disponibile all'URL `https://github.com/iamgio/quarkdown`.

Capitolo 2

Background

La motivazione alla base della creazione di Quarkdown nasce dall'ostica curva di apprendimento ed elevata verbosità e complessità che è possibile incontrare redigendo documenti con LaTeX. Nonostante ciò, bisogna interrogarsi sul motivo per cui tale strumento riesca ad imporsi tra i più utilizzati nel suo ambito, ed inamovibile da decenni. Il vantaggio più grande che LaTeX mette a disposizione è la totale possibilità di alterare un documento a proprio piacimento, lo svantaggio risiede invece nel codice innecessariamente complesso e di difficile lettura che viene richiesto dall'utente per ottenere tali risultati.

Senza grande fortuna nella ricerca di uno strumento che permettesse un punto intermedio tra controllo e usabilità, è iniziata la pianificazione delle fondamenta di Quarkdown con la premessa *"crea tanto, scrivi meno"* ed il quesito: **cosa deve avere questo software in più rispetto alla concorrenza**?

2.1 Analisi dei competitor

Questa sezione analizzerà i principali strumenti presenti nel mercato al fine di analizzare i loro punti forti e lacune per permettere un'adeguata pianificazione delle funzionalità di Quarkdown.
Nella parte finale del documento, in sezione 14.1, saranno messi a confronto questi

stessi strumenti con una versione completa di Quarkdown, mostrando snippet di codice di vario tipo.

LaTeX

> **Esempio 1.** Di seguito si mostra il codice LaTeX richiesto per creare una sezione contenente una sottosezione con una lista non ordinata ed un testo centrato orizzontalmente:

```latex
\section{Sezione}
\subsection{Sottosezione}
\begin{itemize}
    \item \textbf{Primo} elemento
    \item \textbf{Secondo} elemento
\end{itemize}
\begin{center}
    Questo testo si trova al \textit{centro}.
\end{center}
```

Quarkdown nasce con in mente l'usabilità e la facilità di apprendimento per l'utente basando la sua sintassi su quella minimale di Markdown, risultando estremamente familiare a chiunque vi abbia già prodotto artefatti tra cui documentazione open source.

> **Esempio 2.** Il codice precedente assume questa forma in Markdown:

```markdown
# Sezione

## Sottosezione

- **Primo** elemento
- **Secondo** elemento

Questo testo non si trova al centro:
non è possibile in plain Markdown!
```

Si nota facilmente l'ostacolo a cui si giunge scrivendo un documento con Markdown: non viene offerta una personalizzazione della struttura del documento,

in quanto il linguaggio si limita a mostrare il testo che gli viene passato in input formattandolo in base a regole prestabilite. Qualora si volessero applicare personalizzazioni non previste dallo standard, quali gestire l'allineamento di un elemento, l'unica possibilità risiede nell'aggiunta di codice HTML che viene direttamente dato in pasto al browser.

A tal fine Quarkdown trarrebbe vantaggio da ambe le parti, giovando della facilità di utilizzo di Markdown relativamente alla formattazione del contenuto, e del totale controllo della struttura dato da LaTeX grazie alle *funzioni*.

> **Esempio 3.** L'allineamento del testo, impossibile da impostare nell'esempio precedente, si otterrebbe in Quarkdown tramite la funzione `center`:
>
> ```
> # Sezione
>
> ## Sottosezione
>
> - **Primo** elemento
> - **Secondo** elemento
>
> .center
>     Questo testo si trova al _centro_ .
> ```

AsciiDoc

AsciiDoc (`https://asciidoc.org`) è un linguaggio destinato principalmente, come suggerisce il nome, alla creazione di documentazione come wiki e file README, e a tal fine supportato anche da GitHub come alternativa a Markdown.
La sua sintassi proprietaria è ricca di elementi semantici (potenzialmente impegnando le capacità mnemoniche dell'utente durante l'apprendimento) particolarmente utili nella scrittura di documentazione, come spiegazioni passo-passo di snippet di codice. Non consente tuttavia un controllo della struttura e del layout del documento.

Quarto

Quarto (`https://quarto.org`) è un sistema che integra i notebook Jupyter nei suoi documenti, rendendo i blocchi di codice eseguibili. A parte ciò, la sua sintassi utilizza Markdown standard ad eccezione di un blocco *header* contenente i metadati del documento e le informazioni sulla sua paginazione. Seppur faccia parte della famiglia di strumenti simili Quarkdown, con esportazione in presentazioni, libri e articoli, non è da ritenere un 'rivale' a causa della finalità diversa e puramente scientifica.

Typst

Typst (`https://typst.app`) è il principale competitor moderno di LaTeX e consente un elevato livello di personalizzazione, permettendo di configurare l'aspetto di ogni elemento (come paragrafi, titoli e immagini) grazie ad un sofisticato linguaggio di scripting.
La concisione della sintassi (proprietaria) di questo strumento rispetto a quella di LaTeX, la versatilità dei tipi di documenti di output (libri, articoli, lettere, presentazioni e altro) e la libertà di personalizzazione offerta sono state d'ispirazione per l'ideazione di Quarkdown.

MDX

MDX (`https://mdxjs.com`) è un'estensione di Markdown che integra JSX (estensione di JavaScript, sviluppata da React, che consente l'iniezione di codice HTML al suo interno) per permettere lo scripting ed il riuso di componenti grafici.
È uno strumento relativamente semplice rispetto agli altri precedentemente analizzati, e destinato principalmente all'utilizzo come libreria in siti web statici, e non supporta paginazione o controllo del documento.

2.1.1 Conclusioni

Alla luce di quest'analisi, si giunge alla conclusione di voler creare uno strumento con una semplice **sintassi familiare**, compatto, flessibile, versatile, che supporti diversi tipi di documento e che al tempo stesso ne permetta una totale personalizzazione in termini di paginazione, struttura ed estetica.

Capitolo 3

Analisi e pianificazione

3.1 Sintassi

La sintassi di base riprende a pieno quella standard di Markdown, rispettando la quasi totalità dei vincoli, linee guida e edge cases definiti dalle minuziose specifiche CommonMark e GitHub Flavored Markdown.

In aggiunta alle funzionalità standard, vengono introdotti nuovi elementi che si ispirano alla sintassi di base, come blocchi di equazioni matematiche contenuti tra `$`, e che verranno mostrati nel prossimo capitolo.

3.1.1 Chiamate a funzioni

Sintassi

La scelta di sintassi più critica riguarda invece le funzioni: è stato necessario progettare la loro sintassi in modo da essere facilmente e comodamente utilizzabili, ma contemporaneamente senza causare conflitti di sintassi con i costrutti già esistenti. Prima di tutto, bisogna individuare le chiamate a funzioni all'interno del codice di input e distinguerle da testo semplice. Il carattere che univocamente definisce una chiamata è stato scelto essere `.` (punto) posto all'inizio di una parola composta da lettere minuscole e maiuscole. Essendo tale carattere quasi esclusivamente utilizzato al termine delle parole, porlo all'inizio dovrebbe ridurre al minimo i possibili

conflitti, lasciando in ogni caso la possibilità di fare un escape con `\.` garantendo che il punto venga interpretato come un carattere testuale.

Argomenti Individuato il nome della funzione, mancano gli argomenti della chiamata. Sono stati disegnati due tipi di argomenti:

- *Fenced*: ogni argomento è contenuto in parentesi graffe, può estendersi anche su più righe e più argomenti fenced si possono succedere separati da zero o più whitespace.
- *Body*: una funzione può accettare al massimo un solo parametro body, rigorosamente ultimo nella lista dei parametri. Un argomento body viene definito andando sulla riga successiva e indentando ogni riga di 4 spazi o un tab, come avviene nelle liste (capitolo successivo). L'argomento può estendersi su più righe, non deve essere contenuto tra parentesi ed è solitamente l'argomento con il contenuto più esteso, in quanto spesso contiene contenuto Markdown innestato. Argomenti di questo tipo sono accettati solamente dalle chiamate a funzione di *blocco* e non *inline* (paragrafo successivo).

Tutti gli argomenti vengono letti come stringhe, e convertiti al tipo desiderato dalla funzione stessa, che si occupa di gestire gli errori in caso di tipi incompatibili.
I tipi possono essere stringhe (`abc`), numeri (`123`), booleani (`true`/`false`, `yes`/`no`), enumerazioni (`small`, come stringhe ma verificate a tempo di compilazione), Markdown (`Contenuto **Markdown**`) o strutture composte come intervalli (`1..10`).

> **Esempio 4.** Una possibile chiamata a funzione è la seguente:
>
> ```
> .nome {Argomento 1} {Argomento 2}
>     Questo è un **contenuto Markdown**
>     inserito nell'argomento body.
> ```

Un argomento può anche essere *named*, che quindi si riferisce ad uno specifico parametro non per posizione, ma per nome. Il nome viene specificato precedendo l'argomento con il nome e `:` (due punti), senza spazi.

> **Esempio 5.** I *named arguments* rendono la lettura più naturale:
>
> ```
> .multiply {3} by:{4}
> ```

Chiamate di blocco e inline Quando una chiamata è isolata, essa si dice *di blocco* e il suo tipo di ritorno è solitamente un elemento di blocco. Solo questo tipo di chiamata accetta l'argomento *body*.

> **Esempio 6.** Il seguente esempio include una chiamata a funzione di blocco:
>
> ```
> Questo è un paragrafo
>
> .box {Titolo}
>     Contenuto del box
>
> Questo è un altro paragrafo
> ```

Una chiamata è invece *inline* quando si trova all'interno di contenuto in linea, ed il suo tipo di ritorno è solitamente un elemento inline.

> **Esempio 7.** Il seguente esempio include una chiamata a funzione inline:
>
> ```
> Il documento .docname è stato creato da .docauthor.
> ```

Look-up

Le funzioni a cui le chiamate fanno riferimento si trovano nella libreria standard scritta in Kotlin fornita all'interno del progetto, nel modulo `stdlib`. In futuro saranno supportate funzioni definite in progetti di terze parti ed importati, ad esempio, in file JAR, o anche definite dall'utente all'interno del file stesso.

Comportamento

Una funzione può avere diversi ruoli:

- Proprietà: impostazione di una proprietà del documento, senza tipo di ritorno.

```
.docname {Quarkdown}
.aspectratio {4:3}
```

- Struttura: impostazione del layout e aspetto del suo body.

```
.center
    Contenuto centrato
```

- Echo: restituzione di un elemento o contenuto testuale che viene rimpiazzato al posto della funzione durante la compilazione, come una macro.

```
.docname


.csv {file.csv}
```

Ulteriori informazioni, tra cui una lista estesa di funzioni, sono contenute nel file di analisi `quarkdown-syntax.txt` nel branch `analysis`.

3.2 Esempio pratico

In fase di analisi è stato realizzato il file regen.md, presente nel branch `analysis`, in cui sono stati ideati sorgenti che ricreerebbero alcune slide del corso di *Programmazione ad Oggetti* del prof. Viroli. Di seguito alcuni esempi:

Esempio 8.

Figura 3.1: Pagina del titolo.

```
.docname {OOP05: Incapsulamento}
.docauthor {Mirko Viroli}
.aspectratio {4:3}

.colortheme {ThemeName}
.layouttheme {ThemeName}

.footer        <-- L'aspetto è gestito dai temi
    Mirko Viroli (Università di Bologna)

    .docname

    a.a. 2022/2023

    .currentpage / .pagecount
```

Esempio 9.

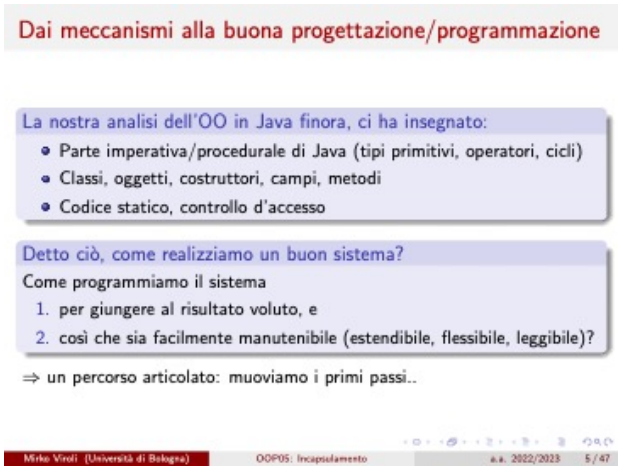


Figura 3.2: Box informativi.

```
.page

# Dai meccanismi alla buona progettazione/programmazione

.box {La nostra analisi}
    - Parte imperativa/procedurale
    - Classi, oggetti, costruttori, campi, metodi
    - Codice statico, controllo d'accesso

.box {Detto ciò, come realizziamo un buon sistema?}
    Come programmiamo il sistema

    1. per giungere al risultato voluto, e
    2. così che sia facilmente manutenibile

$ \Rightarrow $ un percorso articolato:
muoviamo i primi passi...
```

Capitolo 4

Tipi di elementi

4.1 Blocchi

Un blocco è la materia prima su cui si fonda la struttura del documento. Un blocco ha una tipologia ben definita e può contenere altri blocchi a sua volta senza un potenziale limite alla profondità di ricorsione. La seguente immagine mostra un esempio grafico dell'annidazione dei blocchi all'interno di un documento.

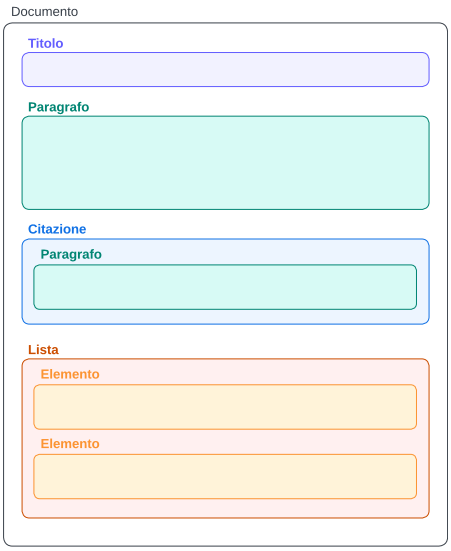


Figura 4.1: Esempio di divisione del contenuto in blocchi

La versione base di Markdown definita dalla specifica CommonMark prevede i seguenti tipi di blocco:

- **Paragraph:** blocco di testo.

```
Questo è il primo paragrafo.

Questo è il secondo paragrafo.
```

- **Heading:** un titolo, preceduto da uno o più `#` in sequenza. La quantità di `#` definisce il livello del titolo;

```
### Questo è un titolo di terzo livello.
```

- **Setext heading:** un titolo, la cui riga successiva contiene una sequenza di `=` per un titolo di primo livello o di `-` per un titolo di secondo livello

```
Questo è un titolo di primo livello.
======
```

- **Block quote:** una citazione, in cui ogni riga inizia per `>`.

```
> Questa è una citazione
> con più righe.
```

- **Fenced block code:** un blocco di codice contenuto tra una sequenza di tre o più ‘ (backtick) o ∼. È inoltre possibile specificare il linguaggio subito dopo la sequenza di apertura.

````
```java
class Point {
    ...
}
```
````

- **Indented block code:** un blocco di codice in cui ogni riga è indentata di 4 spazi o 1 tab.

```
    class Point {
        ...
    }
```

- **Horizontal rule:** un separatore orizzontale dato da almeno tre `-`, `*` o `+` in sequenza.

  ```
  -----
  ```

- **List:** una lista ordinata o non ordinata contenente un numero variabile di elementi. Un elemento può contenere blocchi innestati indentando di 4 spazi o 1 tab. Usare un numero seguito da `.` o `)` rende la lista ordinata, mentre usare un bullet tra `-`, `*` e `+` la rende non ordinata. Una lista richiede lo stesso tipo di bullet per tutti gli elementi, altrimenti si crea una nuova lista.

  ```
  1. Primo elemento
  2. Secondo elemento

      Un paragrafo innestato.

  3. Terzo elemento
  ```

  Nell'estensione GFM, un elemento di una lista può anche contenere un task, che viene poi rappresentato da un checkbox:

  ```
  - [x] Svolto
  - [ ] Non svolto
  ```

- **Link definition:** definizione di un link a cui è possibile fare riferimento tramite l'elemento inline *reference link*. Contiene label a cui fare riferimento, URL e un titolo opzionale.

  ```
  [Quarkdown]: https://github.com/iamgio/quarkdown 'Repository'
  ```

- **HTML:** blocco di codice HTML interpretato direttamente dal browser.

  ```
  <p>Contenuto HTML</p>
  ```

Blocchi introdotti dall'estensione GitHub Flavored Markdown (GFM):

- **Table:** tabella con una riga di intestazione e numero indefinito di righe di contenuto.

```
| Header 1 | Header 2 |
| -------- | -------- |
| Cella 1  | Cella 2  |
| Cella 3  | Cella 4  |
```

Il testo nelle colonne può anche avere un'allineamento. Nel seguente esempio la prima colonna è allineata a destra, mentre la seconda colonna è allineata al centro:

```
| Header 1 | Header 2 |
| -------: | :------: |
| Cella 1  | Cella 2  |
| Cella 3  | Cella 4  |
```

Blocchi introdotti da Quarkdown:

- **Block function:** chiamata a funzione isolata.

```
.nome {Argomento 1} {Argomento 2}
    body
```

- **Math:** blocco di equazione LaTeX.

```
$ x^2 = 4 $
```

Se necessarie più linee, si può adottare una sintassi simile a quella dei *fenced code blocks*:

```
$$$
\begin{cases}
x^2 = 4
y = \infin
\end{cases}
$$$
```

- **Page break:** fine forzata della pagina corrente.

```
<<<
```

4.2 Inline

Gli elementi inline si trovano sempre all'interno di un blocco e succedendosi definiscono il contenuto effettivo del blocco. CommonMark prevede i seguenti tipi di elementi inline:

- **Text:** un semplice contenuto testuale.

```
Questo è del testo.
```

- **Emphasis:** contenuto testuale di lieve importanza.

```
*Emphasis*
_Emphasis_
```

- **Strong:** contenuto testuale di media importanza.

```
**Strong**
__Strong__
```

- **Strong emphasis:** contenuto testuale di alta importanza.

```
***Strong***
___Strong___
```

- **Code span:** frammento di codice in linea.

```
`Code span`
```

- **Link:** collegamento ipertestuale.

```
[Label](url)
[Label](url "Titolo")
```

- **Reference link:** riferimento ad un blocco di *link definition*.

```
[Label][reference]
```

- **Image:** collegamento ad un'immagine.

```
![Label](url)
![Label](url "Titolo")
```

Quarkdown introduce inoltre la seguente sintassi per definire la grandezza di un'immagine (che in standard Markdown è possibile effettuare solo tramite HTML):

```
!(150x100)[Label](...)
!(150x_)[Label](...)
```

dove _ indica una grandezza automatica che mantenga l'aspect ration dell'immagine.

- **Reference image:** immagine che fa riferimento ad un blocco di *link definition*. Anche qui è valida l'estensione della grandezza dell'immagine.

```
![Label][reference]
![reference]
!(150x100)[Label][reference]
```

- **Line break:** interruzione della riga di testo.

```
Linea 1<spazio><spazio>
Linea 2
```

- **Comment:** un commento che è escluso dal rendering.

```
<!-- Commento -->
```

Elementi introdotti dall'estensione GFM:

- **Strikethrough:** contenuto testuale barrato.

```
~~Strikethrough~~
```

Elementi introdotti da Quarkdown:

- **Inline function:** chiamata a funzione unita ad altro contenuto inline.

```
<...> .nome {Argomento 1} <...>
```

- **Inline math:** equazione LaTeX in linea.

```
<...> $ x^2 = 4 $ <...>
```

Capitolo 5

Design

5.1 Pipeline

La compilazione di un linguaggio, che contiene l'intero processo dalla lettura del codice sorgente alla produzione di codice interpretabile da un'altra entità, è composta da operazioni eseguite sequenzialmente e che compongono una pipeline lineare. La pipeline di Quarkdown è così composta:

1. **Registrazione delle librerie**: vengono caricate le librerie (es. `stdlib`) e le loro funzioni sono registrate nel contesto di esecuzione, permettendo un facile look-up.

2. **Lexing:** abbreviazione di *lexical analysis*, si occupa di leggere il codice sorgente ed effettuare una precedura detta di *scansione* o *tokenizzazione*. Qui vengono estratti i simboli, ognuno con una sua tipologia, formando i *token*, piccole porzioni testo che non contengono ancora informazioni dettagliate.

   > **Esempio 10.** In un compilatore Java, data la seguente linea di codice:
   > `String x = "abc";`
   > il lexer produce i token `[String, x, =, "abc", ;]`
   > con rispettivamente tipologie `[type, identifier, assignment, string_literal, end_of_line]`.

3. **Parsing:** il parser ha il compito di analizzare i token prodotti dal lexer e produrre una struttura ad albero detta *Abstract Syntax Tree* (AST), in cui ogni nodo contiene le sue proprietà e tiene traccia dei suoi nodi figli in una lista ordinata.
   Il nodo radice dell'albero è un blocco artificiale chiamato `Document`.

4. **Function call expansion:** messi in coda durante il parsing, i nodi di chiamata a funzione vengono espansi aggiungendo all'AST l'output della chiamata, che fa riferimento ad una funzione registrata nel contesto (altrimenti viene prodotto un errore).

5. **Tree traversal:** l'AST viene interamente percorso dalla radice in modalità depth-first (DFS) al fine di estrarre informazioni utili sulla struttura del documento, come ad esempio per la costruzione del sommario.

6. **Rendering:** l'AST viene percorso nuovamente, traducendo ogni nodo dell'albero in un codice di output in un linguaggio target (HTML, e possibilmente LaTeX in futuro).

7. **Post-rendering:** vengono effettuate operazioni sul codice prodotto dalla fase di rendering, come l'inserimento in un template dedicato ad un libro o ad una presentazione. Seppur non contemplato dalle specifiche, il codice HTML può essere sanificato da elementi potenzialmente pericolosi per la sicurezza o che possono causare malfunzionamenti.

Si disegna di seguito un diagramma di UML per la pianificazione e il collegamento dei singoli elementi che compongono la pipeline. In esso si può notare la natura sequenziale della stessa: l'output di uno stadio i è l'input per lo stadio $i+1$.

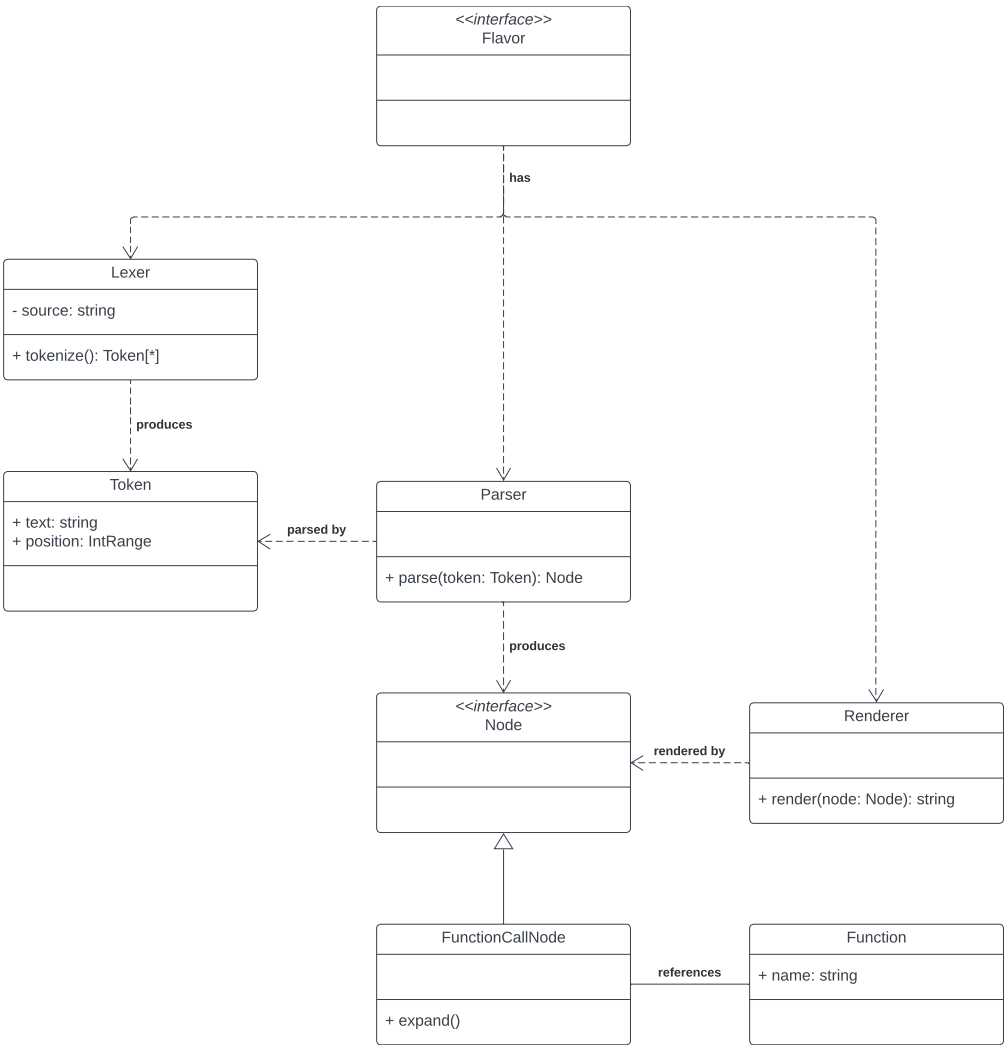


Figura 5.1: Pianificazione del flusso della pipeline.

La novità di Quarkdown rispetto ai flavor tradizionali si trova nella parte bassa del diagramma: un nodo *funzione* è espandibile: eseguita la sua funzione corrispondente, l'output è salvato a sua volta come nodo.

5.2 Flavor

I punti cardine della pipeline si basano su implementazioni concrete che possono dipendere dalle preferenze dell'utente: si vorrebbe, ad esempio, disabilitare le funzionalità avanzate di Quarkdown per limitarsi a scrivere Markdown semplice. Per garantire questa flessibilità sono stati introdotti i `Flavor`, che definiscono le caratteristiche della pipeline.
I flavor supportati sono `BaseMarkdown`, che segue lo standard CommonMark con diverse estensioni del GitHub Flavored Markdown, e `Quarkdown`, che estende `BaseMarkdown` con nuove funzionalità.

Tale struttura dei flavor è rappresentata dal seguente diagramma, che mostra come ogni flavor possa mettere a disposizione le sue componenti della pipeline attraverso pattern di tipo *abstract factory*.

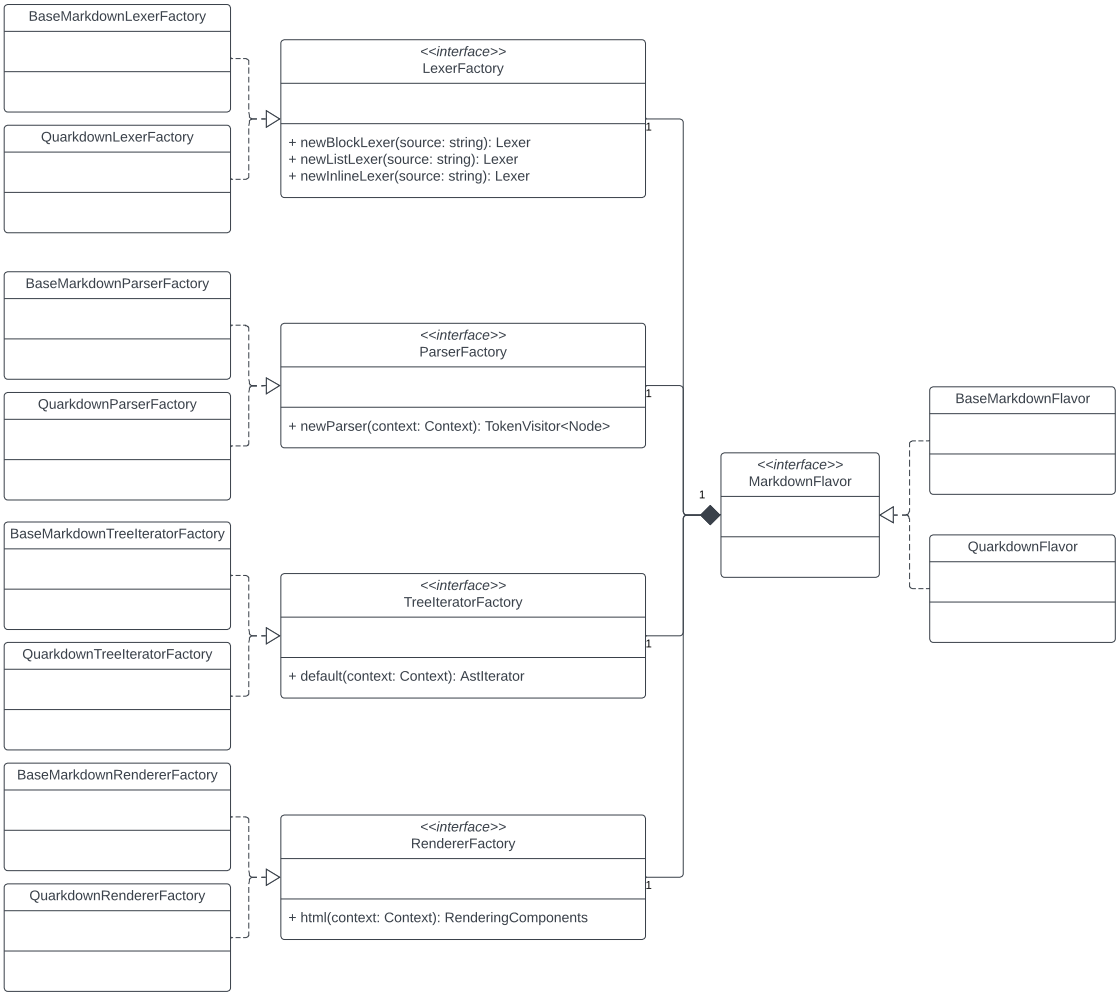


Figura 5.2: Architettura dei flavors.

Quest'architettura permette una discreta estendibilità del programma, garantendo anche la possibilità di far aggiungere nuove funzionalità agli utenti utilizzando Quarkdown come libreria di un progetto terzo.
Le proprietà fornite dai flavor e gli oggetti prodotti dalle loro factory saranno approfonditi nei prossimi capitoli.

Capitolo 6

Implementazione (preambolo)

In questo piccolo capitolo vengono descritti dettagli implementativi che coinvolgono l'intero progetto. I capitoli che seguono trattano rispettivamente uno stadio della pipeline, ognuno diviso nella propria sezione di design e di implementazione.

6.1 Struttura

Il progetto è diviso in sotto-moduli Gradle, ognuno con uno scopo ben preciso e indipendente dagli altri al fine di distinguere con assenza di ambiguità le responsabilità e competenze di ogni componente, salvo dipendenze unidirezionali verso il modulo `core`, e sono:

- `core`: intera logica del programma, dal codice sorgente alla produzione di un artefatto;
- `cli`: interfaccia a riga di comando, che prende il percorso del file di input da argomenti ed effettua logging dei risultati sullo standard output;
- `stdlib`: collezione delle funzioni di base offerte;
- `test`: test che percorrono l'intera pipeline.

6.2 Test automatici

Il testing automatico con `kotlin-test` (basato su JUnit) copre tutte le fasi della pipeline:

1. Lexing: dati in input file Markdown contenenti elementi misti, vengono verificate sequenzialmente le tipologie dei token estratti;
2. Parsing: dati in input file Markdown contenenti lo stesso tipo di elemento, vengono verificate per ogni elemento le proprietà del suo nodo prodotto, che può anche contenere contenuto innestato. Eseguito per ogni tipo di token;
3. Function: chiamate a funzione indipendenti dal documento;
4. Function expansion: chiamate a funzione che influenzano il documento;
5. Rendering: dati nodi in input, viene verificata l'uguaglianza del loro rendering con il contenuto HTML atteso. Eseguito per ogni tipo di nodo;
6. Full pipeline: dato un codice Quarkdown in input, viene verificato il suo output, assicurando quindi una corretta esecuzione di tutte le fasi della pipeline.

Capitolo 7

Lexing

Un lexer si trova alla base di qualunque linguaggio, grande o piccolo che sia. Il nome *lexer*, interscambiabile con *scanner* o *tokenizer*, richiama il concetto di analisi lessicale, ossia la scansione del codice e analisi del suo legame con il vocabolario del linguaggio.
Noi umani usiamo il lexing tutti i giorni senza rendercene conto: le frasi che sentiamo e che leggiamo sono collezioni ordinate di caratteri - è il nostro cervello che analizza il loro contenuto, individuando nomi, verbi, preposizioni e altre strutture grammaticali, che poi dovranno essere legate tra loro per ricavare il significato della frase.

L'azione che un lexer svolge è quella di suddividere un codice sorgente, che non è altro che un array di caratteri, in una lista ordinata di componenti più piccoli detti **token**.
Un token non deve contenere informazioni raffinate, ma solamente il suo contenuto testuale estratto dalla stringa originaria, la sua posizione nel codice sorgente - che risulta utile per mostrare eventuali messaggi di errore - ed il suo tipo.

7.1 Design

In Quarkdown, il tipo di un token è rappresentato da una delle sottoclassi della classe astratta `Token` contenente le informazioni lessicali di base all'interno: testo e posizione. In una prima progettazione il tipo era rappresentato da un'enumerazione come campo `type` di una classe concreta `Token`, ma si è deciso di utilizzare una gerarchia per favorire flessibilità, eleganza e per permettere l'utilizzo del visitor pattern, particolarmente utile nella fase di parsing e approfondito nel prossimo capitolo.
Ad ogni token è inoltre associato un elemento di tipo `TokenData` che contiene il contenuto testuale, la sua posizione di inizio e fine e i suoi gruppi di cattura, che verranno approfonditi più avanti in questo capitolo.

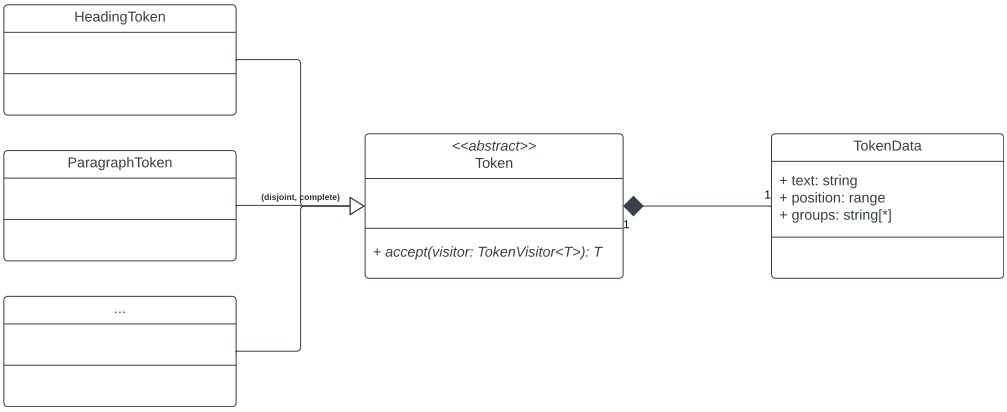


Figura 7.1: Architettura dei token.

7.1.1 Strategia di lexing

Sono diverse le tecniche utilizzabili per estrarre token da una stringa, ma quelle più importanti e adatte al progetto si riconducono a due scuole principali: scansione **carattere per carattere** o estrazione via **regex**.

- La prima tecnica è la più completa e permette assoluta libertà nella scansione di linguaggi complessi, ma al rischio di scrivere codice prolisso e difficilmente mantenibile. Spesso tali lexer sono generati automaticamente da strumenti

come ANTLR e Flex, che espongono allo sviluppatore un'interfaccia più semplice ed astratta.

- La seconda utilizza espressioni regolari (regex) per creare gruppi di cattura ed è adatta a linguaggi - come suggerisce il nome - regolari, ossia la cui grammatica non dipende da un contesto. Permette di avere codice più conciso, delegando la logica di *matching* al motore alla base delle regex, al prezzo di un piccolo, se non impercettibile, calo di performance.

La scelta di quale tecnica adottare per Quarkdown è stata ardua e graduale, rendendo lo sviluppo del lexer una fase ricca di iterazioni di riprogettazione e refactoring per riuscire a comprendere i pro e i contro di ognuna delle due strategie. Il trionfo è stato finalmente trovato dall'approccio tramite regex, assicurando sia un'adeguata flessibilità che un'ottima possibilità di astrazione per poter implementare strategie di lexing differenti senza necessità modificare l'architettura di base.

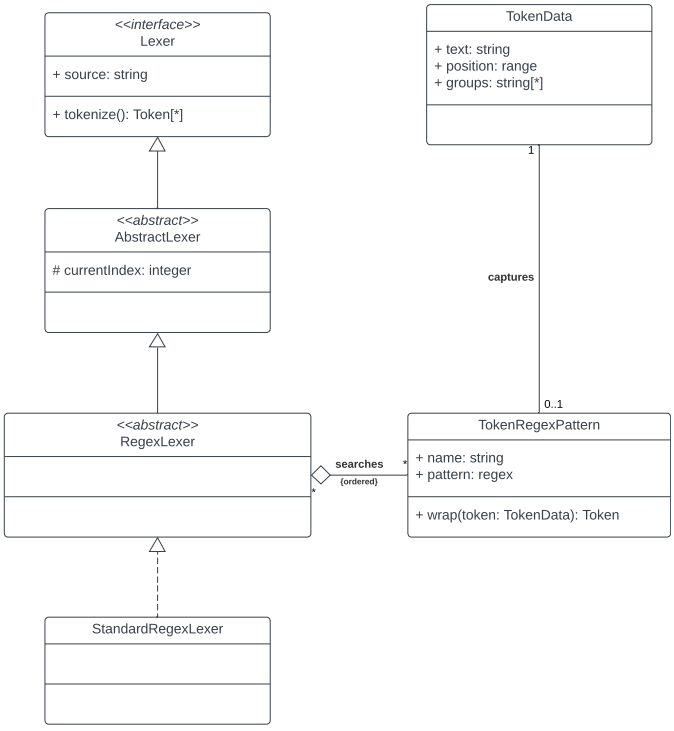


Figura 7.2: Architettura del regex-based lexer.

Un fattore determinante per questa scelta è inoltre stato Marked, il più performante e usato parser Markdown operante su browser, che ho analizzato a fondo per avere una panoramica completa della pipeline. Questo strumento fa un uso intensivo di espressioni regolari testate da milioni di utenti, che ho riutilizzato o manipolato così da avere una base solida e velocizzando i tempi di sviluppo.

7.1.2 Architettura finale

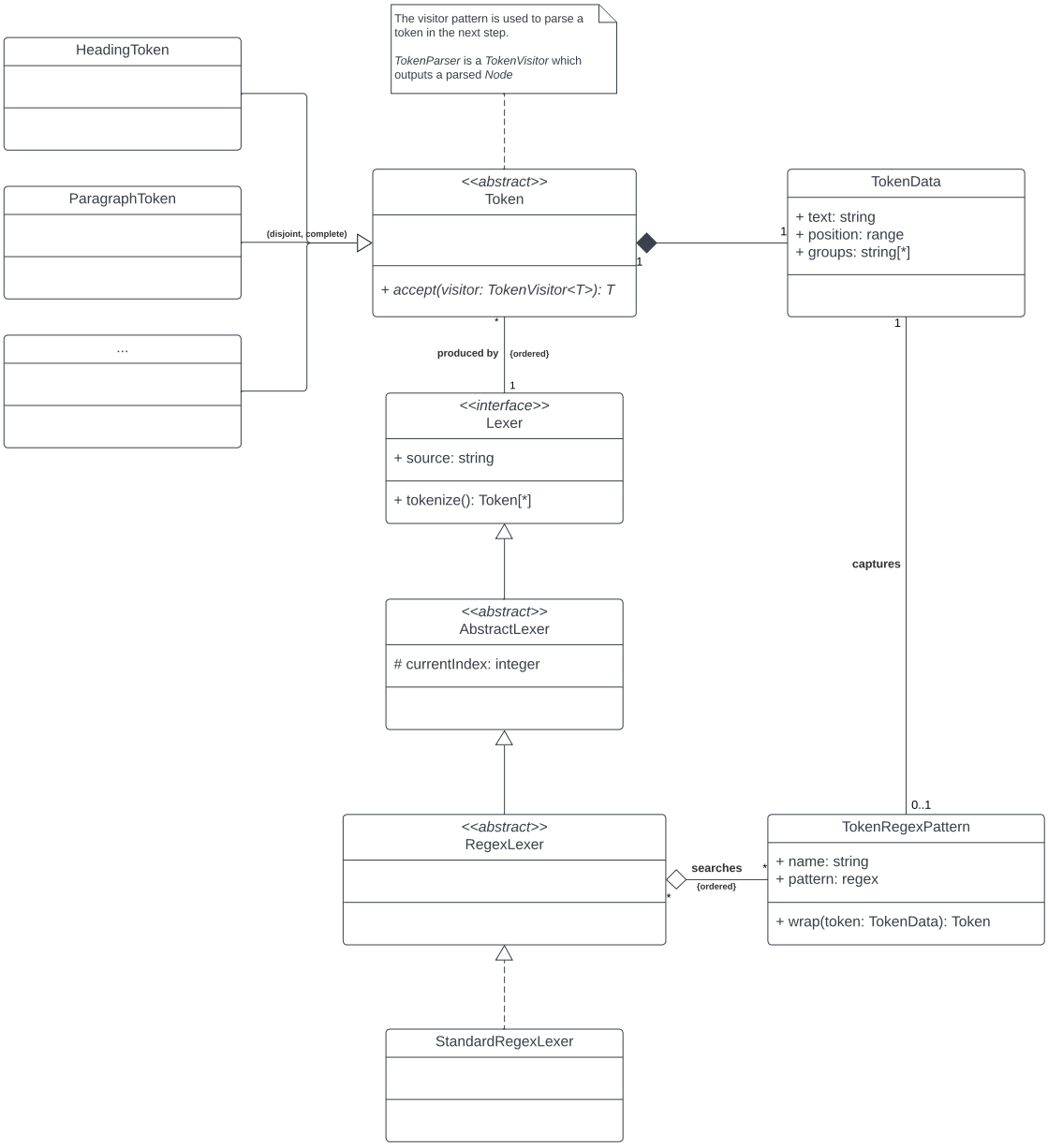


Figura 7.3: Architettura completa del lexer.

7.2 Implementazione

Dati in pasto i pattern a `RegexLexer`, vengono impacchettati in un'unica grande regex dalla seguente forma:

```
(?<GROUPNAME1>expression1)|(?<GROUPNAME2>expression2)|...
```

Ciò ha il grande vantaggio di dividere la stringa di input in catture distinte e mai sovrapposte, mantenendo un concetto di priorità, con i gruppi all'inizio dell'espressione aventi priorità maggiore. Ogni gruppo ha un nome, per cui si parla di *named groups*: per ogni cattura si accede al nome del suo gruppo, ottenendo così il tipo di token a cui quella porzione di testo si riferisce tramite un semplice lookup.
Questo meccanismo permette una discreta divisione delle responsabilità tra la suddivisione della stringa in token e l'identificazione della tipologia per ognuno tramite un'API minimale:

```
val lexer = StandardRegexLexer(
    source,
    listOf(pattern1, pattern2, pattern3, ...),
)
```

con i pattern definiti come:

```
val pattern =
    TokenRegexPattern(
        name = "MyPattern",
        wrap = ::MyToken,
        regex = "regex".toRegex(),
    )
```

con:

- `name` nome del gruppo di cattura, utilizzato per effettuare il lookup;
- `wrap` funzione `TokenData` $\rightarrow$ `Token` che ottiene il sottotipo di `Token` corrispondente al pattern in questione. Si noti che in Kotlin `::Classe` corrisponde a `Classe::new` in Java;

- `regex` espressione che cattura la porzione di testo, estraendo così l'istanza di `TokenData` corrispondente ad ogni match.

Esempio 11. Il pattern che cattura i titoli (da 1 a 6 `#` consecutivi, seguiti da uno spazio e da testo) si presenta in questo modo:

```
val heading =
    TokenRegexPattern(
        name = "Heading",
        wrap = ::HeadingToken,
        regex =
            "^ {0,3}(#{1,6})(?=\\s|$)(.*)(?:\\n+|$)"
                .toRegex(),
    )
```

Spiegazione dell'espressione:

1. `^ {0,3}`: fino a 3 spazi all'inizio della riga, come previsto dalla specifica;
2. `(#{1,6})`: da 1 a 6 `#` consecutivi;
3. `(?=\\s|$)`: la sequenza di `#` deve essere succeduta da un whitespace o dalla fine della riga (nel secondo caso si ha un titolo vuoto);
4. `(.*)`: sequenza di qualunque tipo di carattere;
5. `(?:\\n+|$)`: la sequenza si ferma quando viene incontrata la fine della riga. il `?:` indica di non creare un gruppo di cattura, in quanto l'informazione catturata non è rilevante per il parsing.

Questo è uno dei pattern più semplici presenti nel lexer di Quarkdown. Si trovano diversi pattern che si compongono di altri pattern attraverso un `RegexBuilder` creato ad hoc, che può essere utilizzato, ad esempio, in questo modo:

```
regex =
    RegexBuilder("pattern1|pattern2")
```

```
        .withReference("pattern1", "regex1")
        .withReference("pattern2", "regex2")
        .build()
```

Tutti i pattern utilizzati, per quanto utilizzino espressioni complesse, non hanno contenuti abbastanza rilevanti da essere elencati in questo documento per motivi di brevità. Si basti tenere a mente il loro meccanismo di base che permette di associare a loro il tipo di token corrispondente.

7.2.1 Contenuto innestato

La divisione del contenuto testuale in blocchi è il primo passo da effettuare per comprendere le diverse parti di cui è composto il documento ed attribuire ad ognuna un significato. Come preannunciato, i blocchi possono essere innestati in altri blocchi - quest'implementazione del lexer ignora temporaneamente questo dettaglio, limitandosi a individuare i blocchi di 'primo livello'. L'**elaborazione degli elementi innestati è delegata al parser**, che come vedremo nel prossimo capitolo esegue ricorsivamente l'operazione di lexing sul contenuto del blocco, fino ad effettuare lexing e parsing degli elementi in linea, così generando l'albero dei nodi.

Capitolo 8

Parsing

Una volta ottenuta la lista ordinata dei token dal lexer, il prossimo step prevede di ricavare informazioni più raffinate a partire dai dati crudi offerti da ogni token.

8.1 Design

Alla fine del processo di parsing si otterrà un albero che definisce la struttura del documento analizzato in precedenza dal lexer, in cui ogni nodo corrisponde ad un elemento che può presentare a sua volta elementi figli (`NestableNode`) e/o del contenuto inline (`TextNode`).
Tale albero è noto come *abstract syntax tree* o AST.

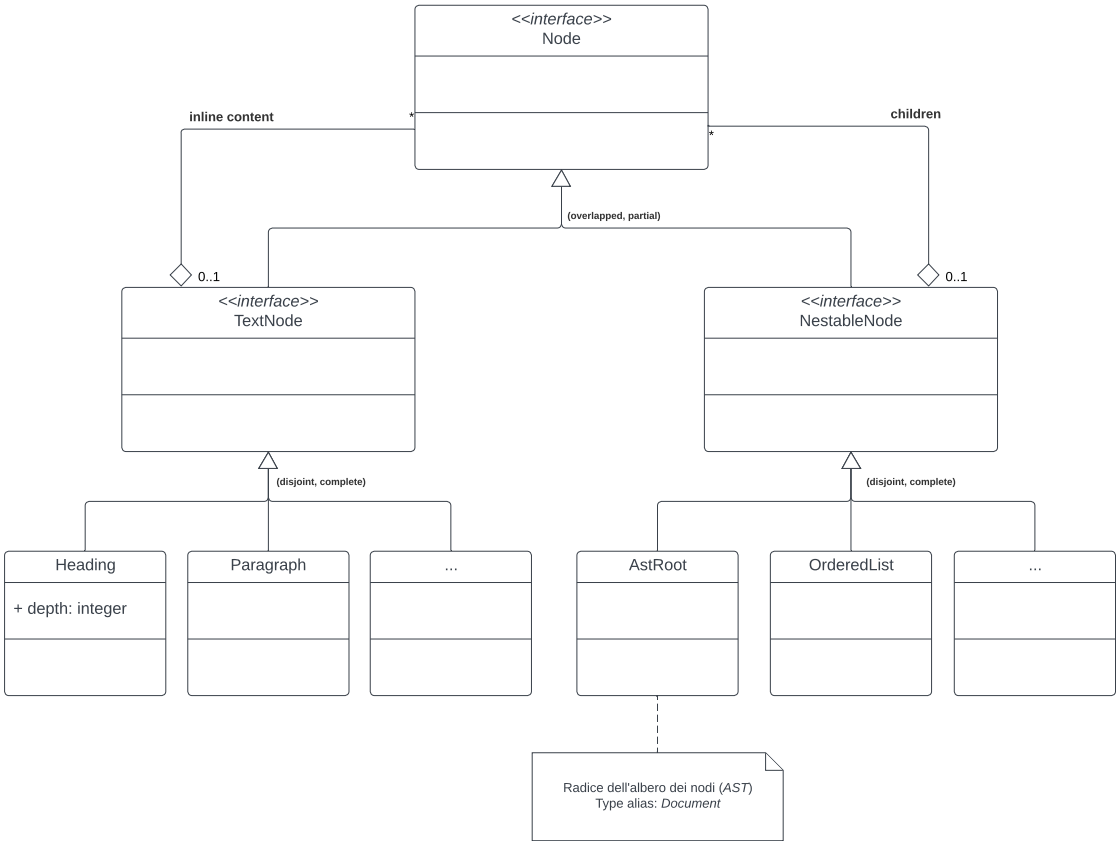


Figura 8.1: Gerarchia dei nodi.

Esempio 12. Dato il seguente input Markdown:

````
# Titolo

Un paragrafo con del testo.

1. Primo elemento della lista.
2. Secondo elemento della lista.
    ```
    Del codice.
    ```
3. Terzo elemento della lista.
````

Ci si attende il seguente AST:

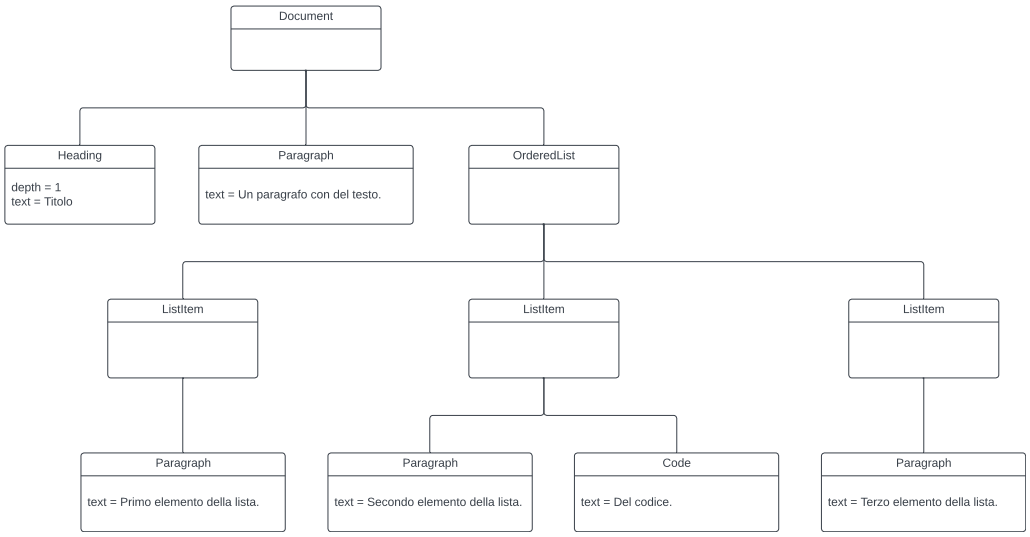


Figura 8.2: AST prodotto dal precedente codice Markdown.

Laddove in un complesso linguaggio di programmazione si possano avere speciali combinazioni di più token per ottenere una determinata funzionalità, in Markdown un token corrisponde *sempre* ad un elemento finale del documento, anche se non sempre visibile (si basti pensare all'importanza delle linee vuote che, pur non avendo un rendering visibile, delimitano inizio e fine tutti i tipi di blocchi).
Si ha dunque un rapporto 1:1 tra numero di token e numero di elementi prodotti, il che apre la strada ad un noto pattern di programmazione object-oriented: il **visitor** pattern.

8.1.1 Visitor pattern

L'importanza del visitor pattern si fonda sulla riduzione del carico di lavoro svolto da ogni specializzazione di una determinata superclasse garantendo compattezza e leggibilità, delegando l'operazione voluta ad un `Visitor` dotato di metodi `visit` - con tanti overload quanti sono le specializzazioni visitabili - e affidando alla superclasse un metodo `accept(Visitor)`, come descritto dal seguente diagramma:

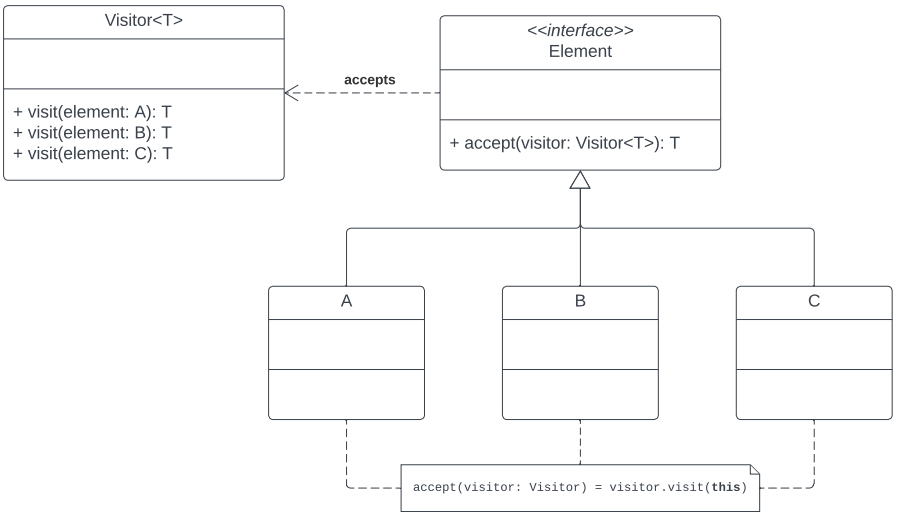


Figura 8.3: Visitor pattern.

8.1.2 Token visitor

Tornando nel nostro dominio, l'obiettivo è quello di generare un elemento a partire da un token, per cui si ha un overload di `visit` per ogni tipo di token risultante della fase di lexing.

Al fine di distinguere le responsabilità e mantenere un codice ordinato, `BlockTokenVisitor` e `InlineTokenVisitor` sono separati per gestire rispettivamente gli elementi di blocco (come titoli, paragrafi e liste) e gli elementi inline (link, grassetto, immagini, ecc.), per poi venire riaccomunati in un'interfaccia `TokenVisitor` che estende entrambi i due tipi di visitor.

I due sotto-visitor sono implementati dai parser `BlockTokenParser` e `InlineTokenParser`, che producono un nodo dell'albero (`Node`) a partire un token. L'interfaccia comune trova invece implementazione nel `TokenVisitorAdapter`, che delega le operazioni di `visit` al sotto-visitor adatto. I collegamenti tra tali componenti sono rappresentati dal seguente diagramma:

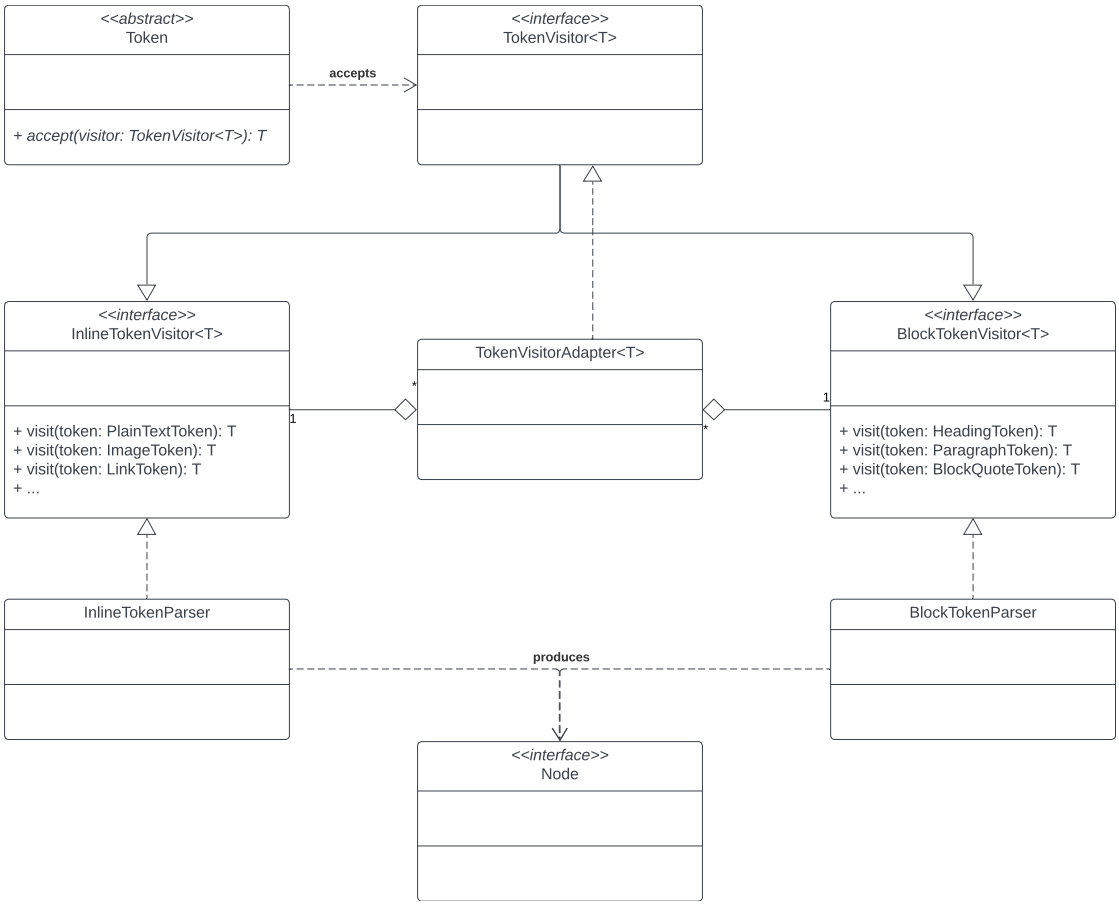


Figura 8.4: Architettura del token visitor.

8.1.3 Architettura finale

Di seguito si mostra l'intera architettura dietro il processo di parsing dei token, composta dai vari componenti visti in questo capitolo. Si può notare il flusso dall'alto verso il basso dell'informazione, in cui il dato crudo (il token) viene gradualmente convertito in informazione processata (il nodo).

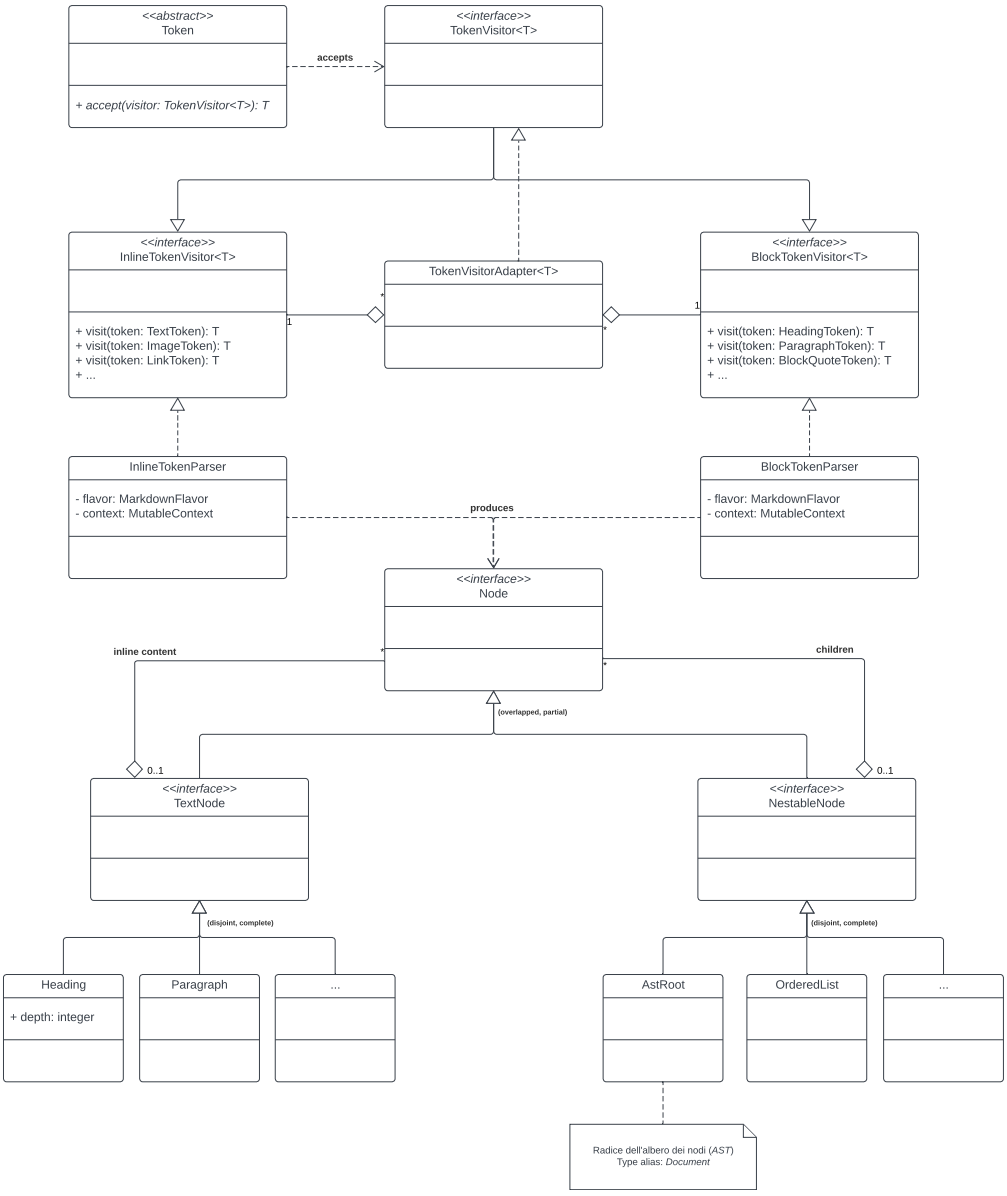


Figura 8.5: Architettura finale del parser.

8.2 Implementazione

Il metodo `accept` è implementato in modo minimale dalle sottoclassi di `Token` per consentire un'adatta implementazione del visitor pattern:

```
class ParagraphToken(data: TokenData) : Token(data) {
    override fun <T> accept(visitor: TokenVisitor<T>) =
        visitor.visit(this)
}

class HeadingToken(data: TokenData) : Token(data) {
    override fun <T> accept(visitor: TokenVisitor<T>) =
        visitor.visit(this)
}

// ...
```

La delegazione del `TokenVisitorAdapter` avviene convenientemente sfruttando la delegazione automatica offerta da Kotlin tramite la keyword `by`:

```
class TokenVisitorAdapter<T>(
    blockVisitor: BlockTokenVisitor<T>,
    inlineVisitor: InlineTokenVisitor<T>,
) : TokenVisitor<T>,
    BlockTokenVisitor<T> by blockVisitor,
    InlineTokenVisitor<T> by inlineVisitor
```

8.2.1 Block token parser

Il `BlockTokenParser` è l'implementazione del `BlockTokenVisitor` che si occupa di definire la struttura del documento in termini di elementi di blocco.

> **Esempio 13.** Di seguito il parsing di un paragrafo, che consiste nel semplice passaggio dal token al nodo del suo contenuto testuale:
>
> ```
> override fun visit(token: ParagraphToken): Node {
>     return Paragraph(text = token.data.text.trim())
> }
> ```

È nel parser che si sfrutta a pieno l'implementazione basata su regex del lexer, consentendo di accedere direttamente ai gruppi di cattura, salvati nel campo `groups` di `TokenData`.

> **Esempio 14.** Si consideri la seguente regex che cattura qualunque numero, opzionalmente preceduto da `+` o `-`:
>
> `[+-]?(\d+)`
>
> 1. `[+-]?` indica la possibile presenza di uno tra i due caratteri;
> 2. `\d+` indica una sequenza continua di cifre.
>
> Inserendo `\d+` tra parentesi tonde si crea un nuovo gruppo di cattura dedicato a quella singola porzione di stringa: data in input la stringa `-15`, sarà possibile ricavare il contenuto numerico `15` accedendo al primo gruppo di cattura.

Visto l'utilizzo effimero di ogni gruppo di cattura, che viene letto solamente una volta, trasformato e assegnato al nodo finale, si utilizza un iteratore per scorrere i gruppi desiderati.

Esempio 15. Si consideri il seguente blocco di codice:

````
```java
public class Point {
    ...
}
```
````

Questo blocco è composto da quattro parti:

1. Il pattern di apertura: ‘‘‘, detto *fence* (recinzione);
2. Il linguaggio dello snippet, in questo caso `java`, che è opzionale;
3. Il contenuto del codice;
4. Il pattern di chiusura, uguale a quello di apertura.

Ognuna di queste componenti è associata ad un distinto gruppo di cattura, compattando il parsing, apparentemente complesso, in poche righe di codice:

```
override fun visit(token: FencesCodeToken): Node {
    val groups = token.data.groups.iterator(consumeAmount = 1)
    return Code(
        language = groups.next().takeIf { it.isNotBlank() }
                        ?.trim(),
        content = groups.next().trim(),
    )
}
```

Analizzandolo si nota:

1. L’ottenimento della sequenza dei gruppi di cattura e la sua trasformazione in iteratore, da cui viene consumato il primo elemento (il pattern di apertura, che è irrilevante);

2. Il secondo gruppo contiene il linguaggio: avvalendosi del comodo metodo di Kotlin `takeIf` si determina il linguaggio del codice se è stato specificato, o `null` in caso contrario;

3. Il terzo gruppo contiene il contenuto del codice.

Problema Paragrafi, titoli e blocchi di codice sono solo alcuni dei blocchi *semplici* di Markdown, ossia senza figli, che non si estendono verticalmente nell'albero dei nodi. È quindi necessario studiare una strategia per elaborare le informazioni contenute in potenzialmente infiniti livelli di profondità di un nodo più complesso rappresentato dall'interfaccia `NestableNode`.
Ogni blocco complesso ha una propria condizione da soddisfare affinché il contenuto testuale su una nuova riga appartenga ad esso. Ad esempio una citazione richiede un `>` all'inizio della riga ed una lista richiede dell'indentazione.

Esempio 16. Si consideri il seguente blocco di citazione:

```
> Primo paragrafo di una citazione.
>
> Secondo paragrafo di una citazione.
>
>> Una citazione innestata.
```

Il risultato atteso del parsing è:

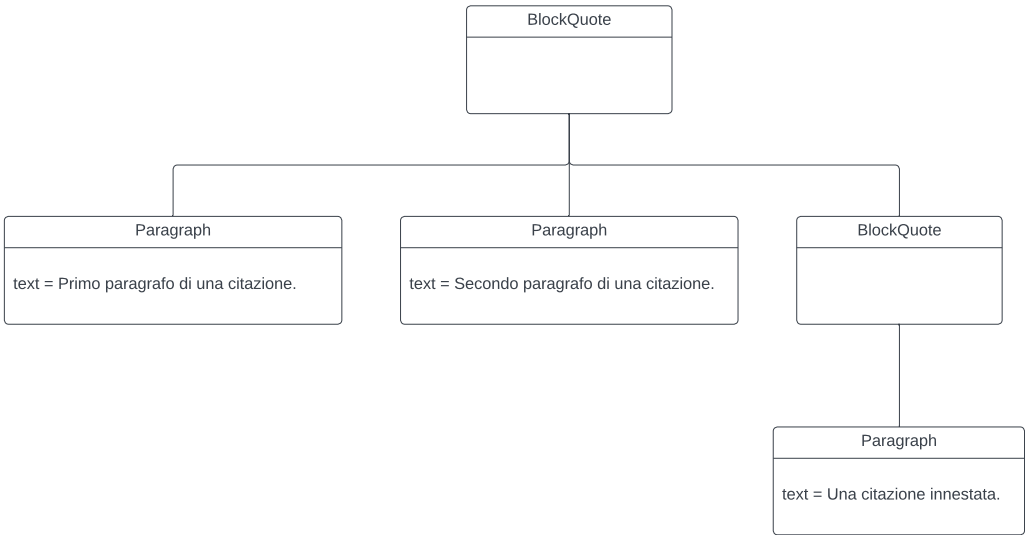


Figura 8.6: AST prodotto dalla precedente citazione.

Soluzione È stato possibile ottenere il sotto-albero di un nodo applicando un parsing ricorsivo: ottenuto il contenuto testuale dell'intero blocco si consumano le condizioni di continuità (ad esempio per una citazione si eliminano i > iniziali) e si procede a rieseguire il processo di lexing e parsing relativo a quel sotto-contenuto fino a raggiungere i nodi foglia.

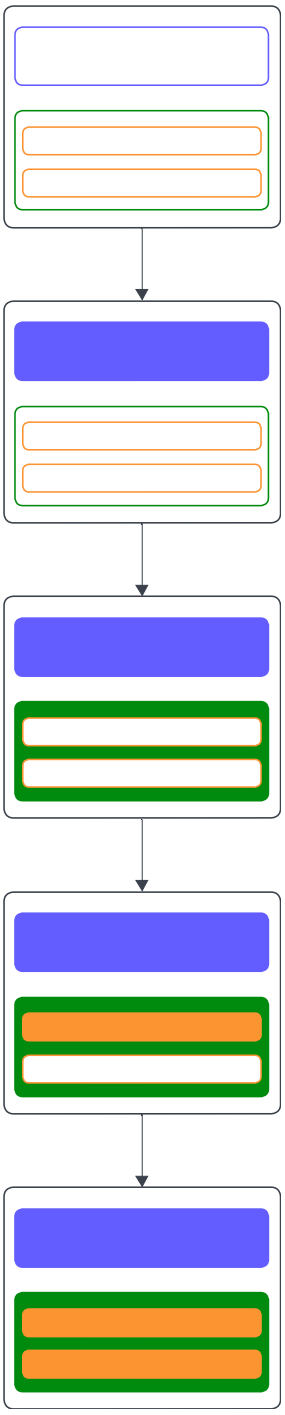


Figura 8.7: Esempio visivo del parsing ricorsivo.

Dal momento che esistono diversi tipi di flavor, e quindi di lexer, il parser tiene memoria del lexer utilizzato in partenza generando nuovi lexer e parser attraverso rispettivamente la `LexerFactory` e la `ParserFactory` del flavor.

```
class BlockTokenParser(private val flavor: MarkdownFlavor)
    : BlockTokenVisitor<Node> {
    ...
    override fun visit(token: BlockQuoteToken): Node {
        // Rimuove i > iniziali tramite regex
        val text = token.data.text
            .replace("^ *>[ \\t]?".toRegex(RegexOption.MULTILINE), "")
            .trim()

        return BlockQuote(
            children = flavor.lexerFactory.newBlockLexer(source = text)
                .tokenize()
                .acceptAll(flavor.parserFactory.newParser()),
        )
    }
    ...
}
```

Dove `acceptAll()` è un'extension function definita come:

```
fun <T> Iterable<Token>.acceptAll(visitor: TokenVisitor<T>): List<T> =
    this.map { it.accept(visitor) }
```

Risultato

> **Esempio 17.** Dato il codice sorgente dello scorso esempio:
>
> ```
> # Titolo
>
> Un paragrafo con del testo.
>
> 1. Primo elemento.
> 2. Secondo elemento.
>     ```
>   Del codice.
> ```

````
    ```
3. Terzo elemento.
````

Eseguendo sequenzialmente lexing e parsing in modalità debug, tramite `-Dloglevel=debug`, si ottiene il seguente output a seguito di una DFS sull'albero:

```
[DEBUG] AST:
AstRoot
        Heading(depth=1, text=Titolo)
        Paragraph(text=Un paragrafo con del testo.)
        Newline
        OrderedList(startIndex=1, isLoose=false)
                BaseListItem
                        Paragraph(text=Primo elemento.)
                BaseListItem
                        Paragraph(text=Secondo elemento.)
                        Code(text=Del codice., language=null)
                BaseListItem
                        Paragraph(text=Terzo elemento.)
```

8.2.2 Inline token parser

Fino ad ora il testo di paragrafi e titoli è stato rappresentato da una stringa, ma ciò non basta per contenere la formattazione del testo data dagli elementi inline. In fin dei conti, un elemento inline non è altro che un nodo dell'albero, nel ramo di un blocco.
L'implementazione dell'`InlineTokenParser` segue la medesima logica del `BlockTokenParser` e supporta anch'essa il parsing ricorsivo.

Basta a questo punto cambiare la definizione dell'interfaccia `TextNode`, implementata da paragrafi e titoli, in:

```
typealias InlineContent = List<Node>

interface TextNode : Node {
    // Prima: val text: String
```
```

```
    val text: InlineContent
}
```

E aggiornando il parsing del testo in `BlockTokenParser`:

```
...

/**
 * @return [this] raw string tokenized and parsed into processed
 *         inline content, based on this [flavor]'s specifics
 */
private fun String.toInline(): InlineContent =
    flavor.lexerFactory.newInlineLexer(this)
        .tokenize()
        .acceptAll(flavor.parserFactory.newParser())

...

override fun visit(token: ParagraphToken): Node {
    return Paragraph(
        text = token.data.text.trim().toInline(),
    )
}

...
```

Risultato

Esempio 18. Dato il codice sorgente:

````
# [Titolo](https://github.com/iamgio/quarkdown)

Del _testo_ .

1. **Primo** elemento della lista.
2. **Secondo** elemento della lista.
    ```
    Del codice.
    ```
3. **Terzo** elemento della lista.
````

Si ottiene il seguente output (riformattato per migliorare la leggibilità):

```
[DEBUG] AST:
AstRoot
    Heading(depth=1, text=[
        Link(
            label=[PlainText(text=Titolo)],
            url=https://github.com/iamgio/quarkdown,
            title=null
        )
    ])

    Paragraph(text=[
        PlainText(text=Del )
        Emphasis
            PlainText(text=testo)
        PlainText(text=.)
    ])

    Newline

    OrderedList(startIndex=1, isLoose=false)
        BaseListItem
            Paragraph(text=[
```

```
            Strong
                PlainText(text=Primo)
            PlainText(text= elemento della lista.)
        ])
    BaseListItem
        Paragraph(text=[
            Strong
                PlainText(text=Secondo)
            PlainText(text= elemento della lista.)
        ])
        Code(content=Del codice., language=null)
    BaseListItem
        Paragraph(text=[
            Strong
                PlainText(text=Terzo)
            PlainText(text= elemento della lista.)
        ])
```

Capitolo 9

Scripting

Questo capitolo tratterà, in maniera sintetica data la complessità, l'argomento delle funzioni e delle chiamate ad esse, tratto caratterizzante di Quarkdown rispetto agli altri flavor di Markdown.
Le funzioni sono sufficienti a rendere lo scripting di Quarkdown **Turing complete**, permettendo la realizzazione di script anche complessi.

9.1 Design

Come preannunciato, un nodo di chiamata a funzione `FunctionCallNode` è costituito da:

- Nome;
- Argomenti fenced (tra parentesi graffe);
- Argomento body (se il nodo è di blocco anziché inline).

Grazie al nome della funzione si effettua un look-up sul contesto di esecuzione per ricavare la funzione a cui fare riferimento e, se non esiste, viene prodotto un errore. Al momento è concessa solo una funzione per uno stesso nome, ma è possibile definire parametri opzionali.

Valori Non tutti i tipi di valori sono adatti ad una funzione Quarkdown: per permettere questo controllo si adopera il wrapper `Value` le cui implementazioni

definiscono i tipi di valori accettati, come `StringValue`, `NumberValue` e molti altri. È inoltre possibile che un tipo sia adatto solamente ad output di una funzione e non ad input in un argomento (basti pensare a `void`), realizzando dunque un'ulteriore gerarchia per distinguere tipi di input da quelli di output. Questo aspetto verrà approfondito in seguito.

Output Ottenuto il valore, racchiuso in `OutputValue`, restituito da una funzione chiamata, esso va trasformato in modo da essere visualizzato nel documento finale. Ad esempio una stringa o un numero possono venire convertiti in un `Text`, un booleano in un `Checkbox` non interattivo, ed un `Node` ritornato è mappato in sé stesso. Anche qui è di nostro aiuto un `Visitor` specializzato che permette la mappatura da una sottoclasse di `OutputValue` ad uno specifico `Node`.
L'accodamento nell'AST dei nuovi nodi prodotti è chiamato *expansion*.

Primo prototipo Un primo diagramma di progettazione ha il seguente aspetto. Si nota come una `FunctionCall` faccia sempre riferimento ad una ed una sola `Function`, invocabile, ed un `FunctionCallArgument` ad uno ed un solo `FunctionParameter`. Un valore (`Value`) può inoltre essere fornito da una `ValueFactory` a partire da stringhe grezze (ad esempio, `ValueFactory#number("42")` → `NumberValue(42)`).

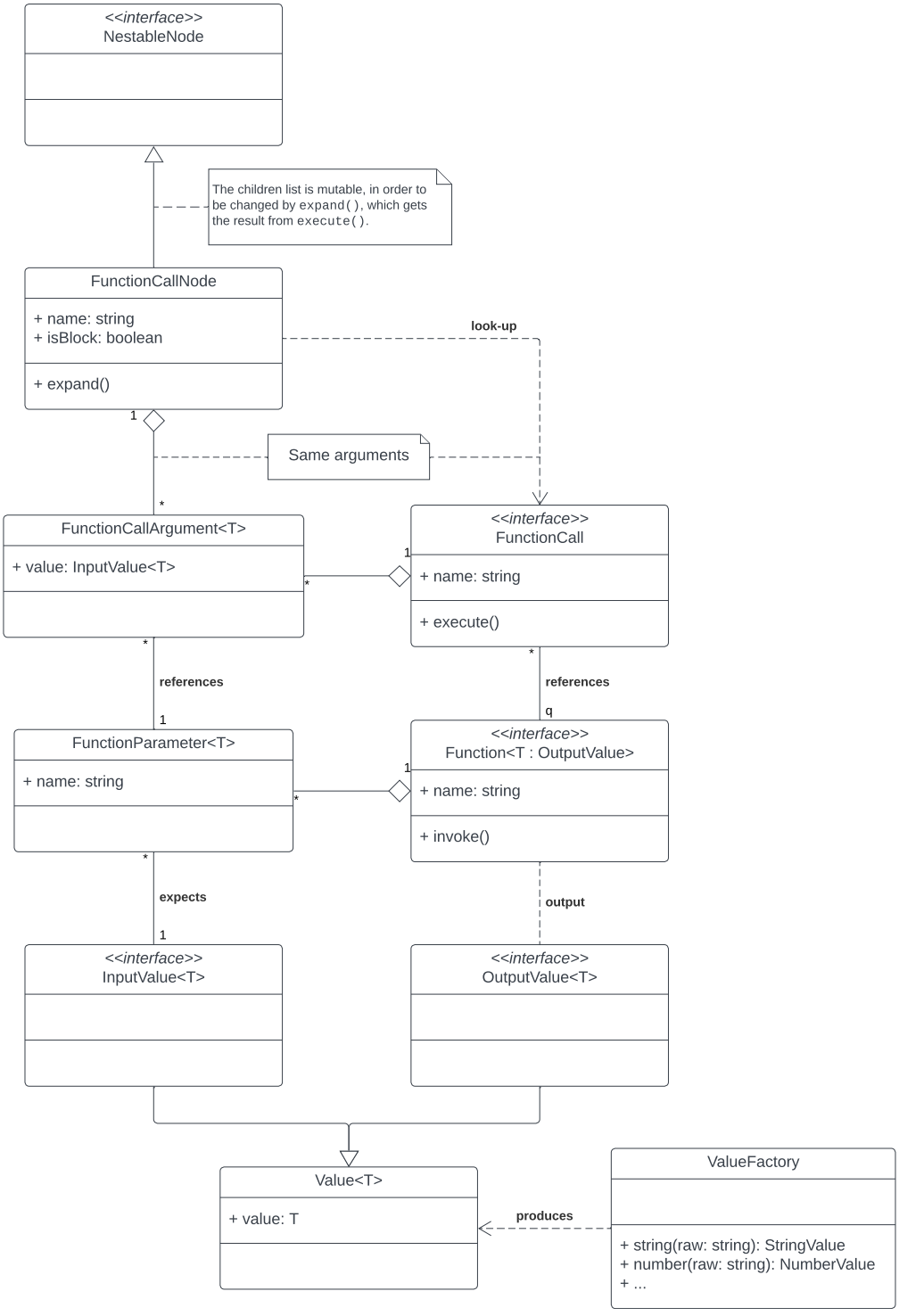


Figura 9.1: Diagramma di progettazione delle funzioni.

9.1.1 Espressioni e valori

Il tema dei valori di input e output, trattato superficialmente in precedenza, è in realtà complicato. Considerando il seguente esempio che esegue l'operazione $3 + (2 + 1)$:

```
.sum {3} {.sum {2} {1}}
```

Il primo argomento, 3, è banalmente convertito in un `NumberValue` dal valore predefinito e statico. La situazione è più articolata nel secondo argomento, in quanto è necessario chiamare la funzione innestata, ottenere il suo output e passarlo in input alla funzione esterna. Bisogna dunque eseguire un'*evaluation* della chiamata a funzione, che in linea più astratta appartiene alla classe delle *espressioni*.

Un'espressione è una qualunque entità convertibile (tramite *evaluation*) in un `Value` statico. L'evaluation è eseguita da un `EvalExpressionVisitor`.
Ai sottotipi di `Expression` appartengono:

- **Chiamate a funzione:** il risultato dell'evaluation è l'output della chiamata;
- **Valori statici (`Value`):** il risultato è il valore stesso;
- **Espressioni composte:** wrapper di una collezione ordinata di espressioni; tali sotto-espressioni vengono unite in un'unica espressione tramite un meccanismo di *appending* (tramite un `AppendExpressionVisitor`) ed il risultato è l'evaluation di quest'espressione.

  > **Esempio 19.** L'*append* di stringhe è la semplice concatenazione di esse: supponiamo l'esistenza di una funzione `.greet` che ritorni la stringa `Hi, <primo argomento>!`. Allora la seguente chiamata: `.greet {welcome back .docauthor}` produrrebbe l'output `Hi, welcome back Giorgio!`. In questo caso l'argomento contiene l'espressione composta da:
  >
  > 1. `welcome back`: una stringa dal valore statico;
  > 2. `.docauthor`: una chiamata a funzione

> Eseguita la funzione e ottenuta la stringa `Giorgio`, essa è concatenata alla stringa statica precedente, producendo un unico `StringValue`.

La distinzione tra valori statici e chiamate a funzione all'interno degli argomenti viene eseguita durante il parsing dei `FunctionCallNode` tramite lexing ricorsivo, in cui vengono distinti i `PlainTextToken` dai `FunctionCallToken`.

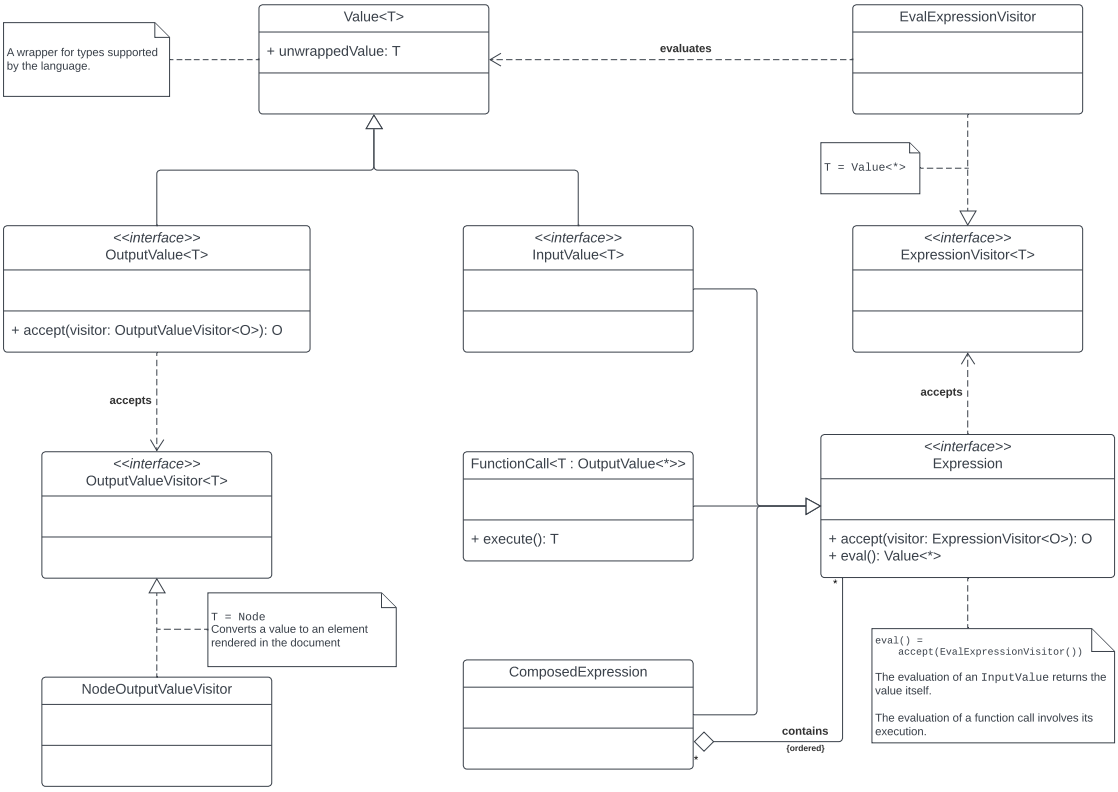


Figura 9.2: Architettura delle espressioni.

I tipi di valori supportati sono riportati nel seguente diagramma di Venn:

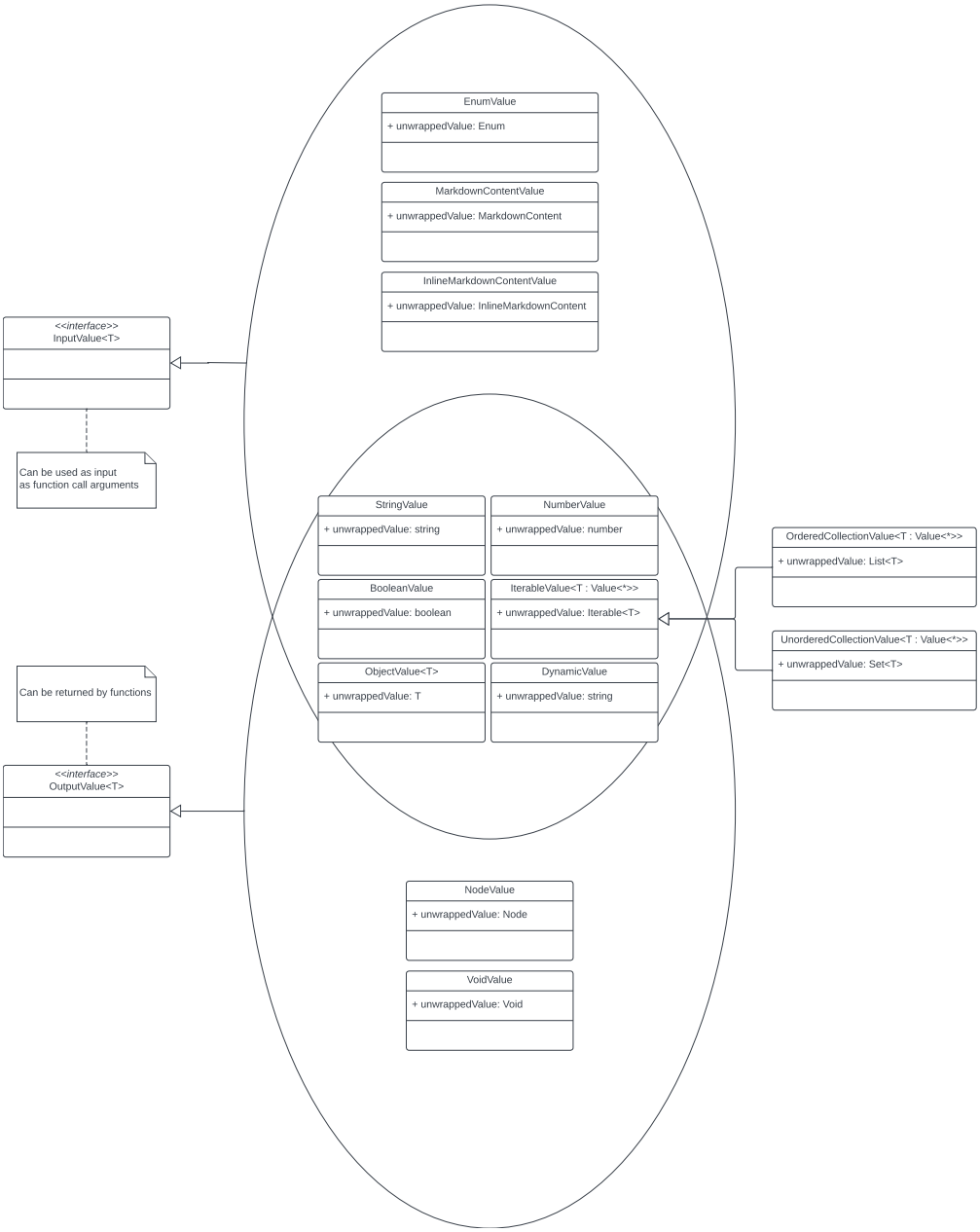


Figura 9.3: Diagramma di Venn dei valori di input (ovale in alto) e output (ovale in basso) supportati.

9.1.2 Chiamate

L'architettura delle chiamate a funzione è del tutto indipendente dall'AST, da cui dipende solamente l'espansione dei nodi, e si mostra nel seguente modo:

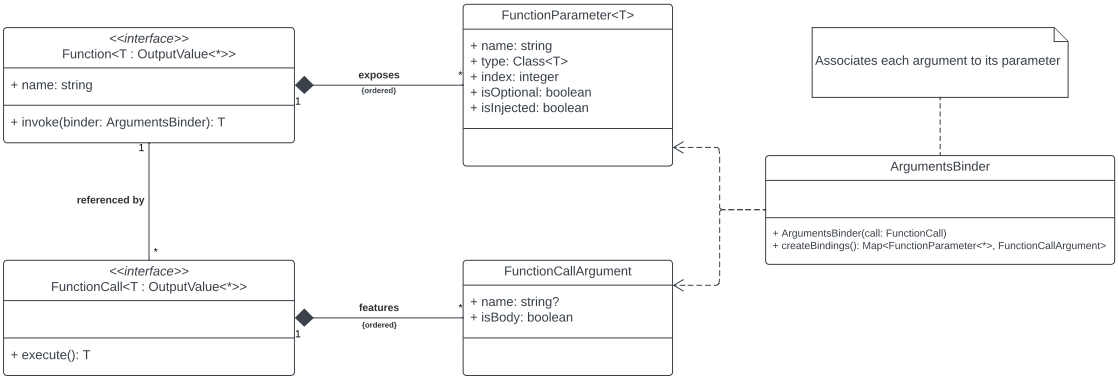


Figura 9.4: Architettura delle chiamate a funzione.

L'`ArgumentsBinder` si occupa di:

1. Associare di ogni parametro della funzione all'argomento passato dall'utente, sulla base di criteri come posizione, opzionalità e nome (nel caso dei *named arguments*);

2. Tentare la conversione di ogni argomento, che si trova in forma di stringa e contenuto in un `DynamicValue`, nel tipo del parametro corrispondente attraverso la static factory `ValueFactory`;

3. Produrre errori in caso di numero errato di argomenti o di incompatibilità tra il tipo atteso e quello ricevuto.

9.1.3 Definizione

L'obiettivo è quello di definire le funzioni in modo naturale sotto forma di funzioni Kotlin. È quindi necessario rappresentare una funzione Kotlin (`KFunction`, gestita dalla libreria `kotlin-reflect`) come una funzione Quarkdown (`Function`), e ciò è reso possibile grazie all'`Adapter` pattern, esponendo i suoi parametri come parametri Quarkdown e permettendo l'invocazione sulla base di un `ArgumentsBinder`.

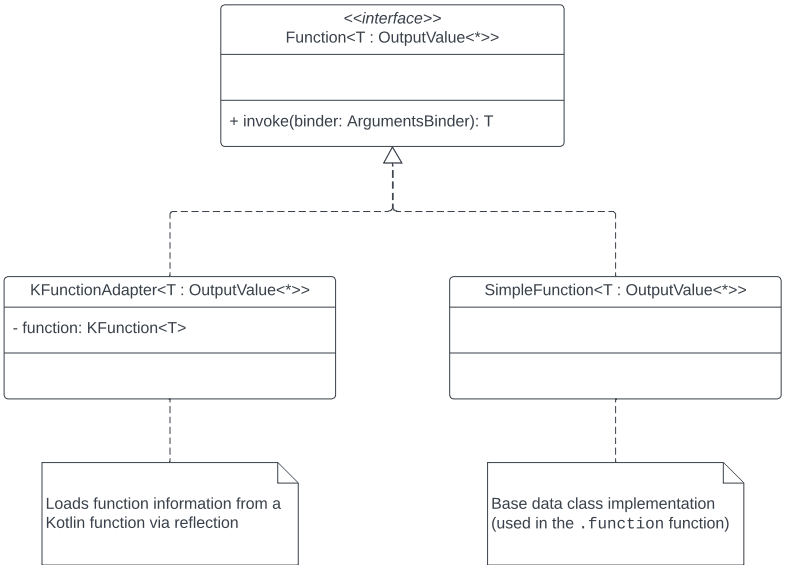


Figura 9.5: Implementazioni di Function

`SimpleFunction` è invece utilizzata per la definizione di funzioni create dall'utente, e verrà trattata alla fine del capitolo.

9.1.4 Librerie e registrazione

Definire una funzione non basta per poterla invocare da un sorgente Quarkdown: è prima necessario registrarla nel **contesto**, ossia l'ambiente di esecuzione condiviso in tutta la pipeline, per permettere il look-up da un'unica fonte. Prima di tutto si vogliono distinguere i vari pacchetti di funzioni, che in Quarkdown sono detti *librerie*: al momento viene incluso automaticamente il modulo interno `stdlib`, ma in futuro verranno supportate anche librerie di terze parti.

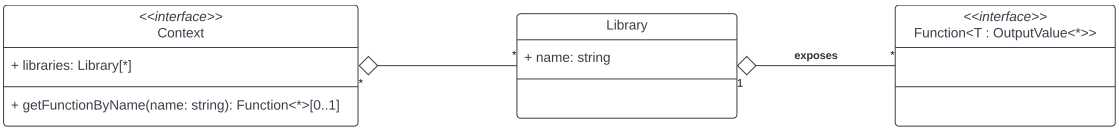


Figura 9.6: Librerie nel contesto.

Ogni libreria espone il suo insieme di funzioni e può essere registrata in `Context`, così favorendo il suo metodo `getFunctionByName(name)`.

Prendendo il modulo `stdlib` come esempio, la sua libreria viene *esportata* dal singleton `Stdlib` dopo aver caricato le funzioni nella maniera voluta. Il caricamento è delegato allo strategy `LibraryLoader<S>`, con `S` *sorgente* da cui estrarre funzioni, e a `MultiLibraryLoader<S>` che usa un *proxy* per estrarre un'unica da più sorgenti dello stesso tipo.

Più nello specifico, `FunctionLibraryLoader` carica una libreria da *una* singola funzione, a cui `MultiFunctionLibraryLoader` fa riferimento per poi impacchettare tutte le micro-librerie insieme in un'unica grande libreria, in questo caso dal nome `stdlib`

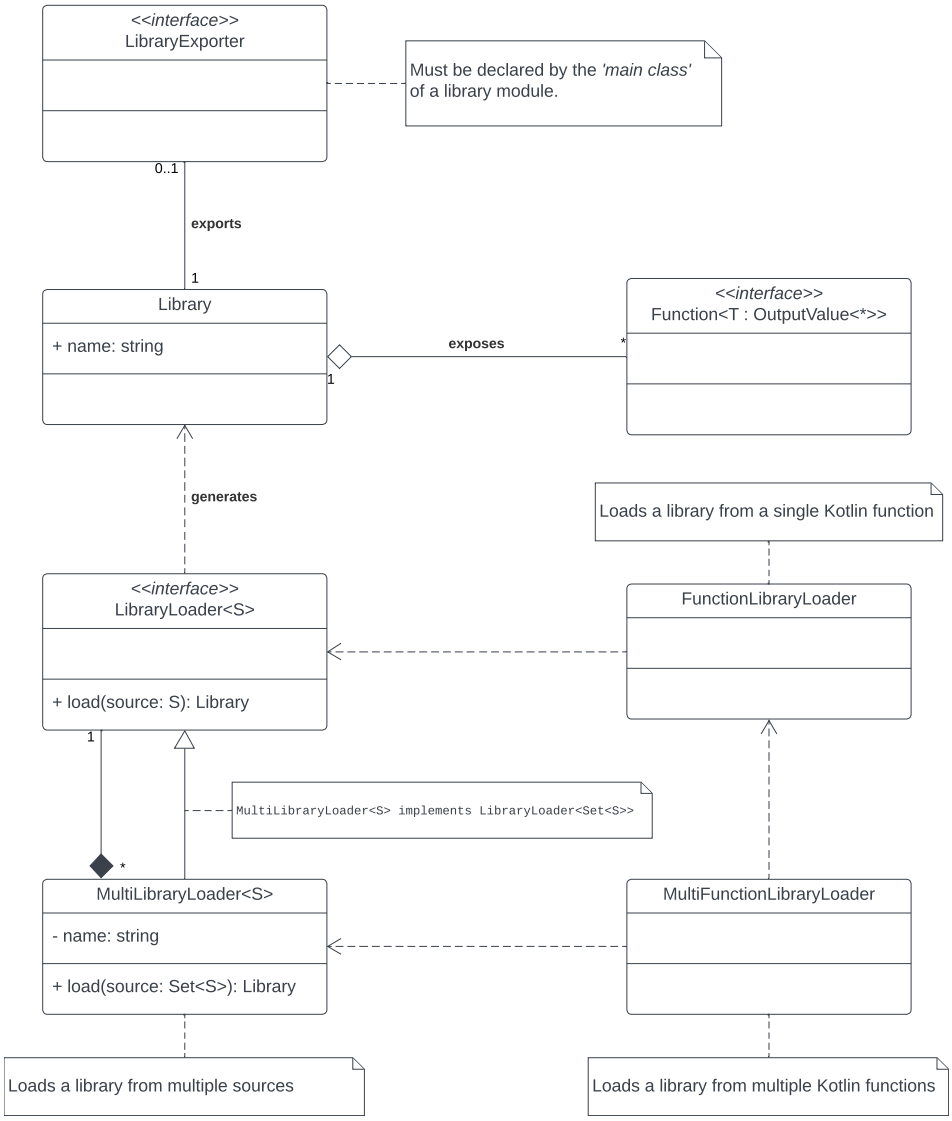


Figura 9.7: Architettura dell'esportazione di librerie.

La libreria è poi acceduta dal modulo `cli` in fase di creazione della pipeline. Le librerie passate alla pipeline vengono registrate automaticamente nel contesto durante la sua inizializzazione.

```
Pipeline(
    ...
    libraries = setOf(Stdlib.library),
    ...
)
```

9.1.5 Espansione

Creata l'architettura che permette di eseguire funzioni in modo indipendente, resta da fare in modo che l'output venga tradotto in un componente grafico visibile nel documento finale, e quindi nell'AST.

Mutabilità A differenza di qualunque altra sottoclasse di `NestableNode` presente nel progetto, `FunctionCallNode` è l'unica il cui attributo `children` è mutabile, e quindi modificabile anche dopo la sua creazione. Questo perché, inizialmente vuoto, il suo contenuto viene popolato dopo l'esecuzione della funzione e quindi la produzione di un output.

Il seguente diagramma mostra i collegamenti tra le varie componenti che gestiscono l'espansione dei nodi funzione. Si può notare la presenza di `UncheckedFunctionCall`: la differenza tra `resolve` e `resolveUnchecked` è che la seconda non controlla l'esistenza della funzione immediatamente, ma in modo *lazy* al momento dell'esecuzione.

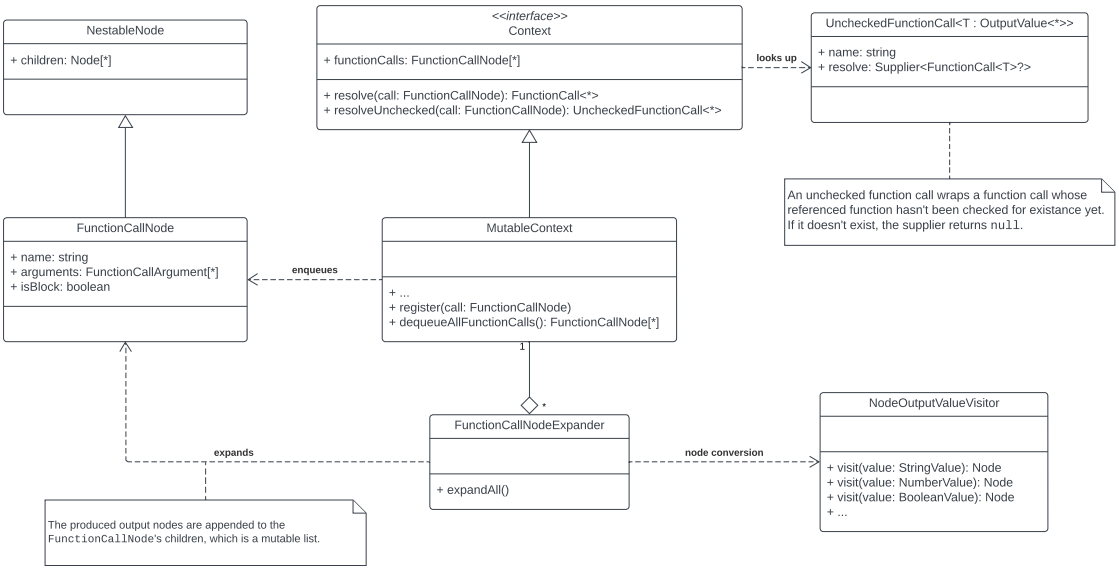


Figura 9.8: Architettura dell'espansione di funzioni.

9.1.6 Gestione degli errori

Markdown non presenta la possibilità di avere errori di compilazione: si può solo incontrare un rendering non conforme alle aspettative dell'utente. Questo cambia però con le funzioni: l'utente deve sapere il motivo per cui una chiamata non sia andata a buon fine.

Un generico errore è rappresentato da `PipelineException`, con le sue sottoclassi che rappresentano specifiche istanze, ad esempio:

- `InvalidArgumentCountException`, `MismatchingArgumentTypeException`: lanciati dal `FunctionArgumentsLinker` nel caso in cui argomenti e parametri non possano essere associati correttamente;
- `UnresolvedReferenceException`: lanciata se una funzione chiamata non esiste nel contesto;
- `FunctionRuntimeException`: lanciata in caso di errore durante l'esecuzione di una funzione.

Tramite riga di comando l'utente può scegliere il tipo di error handling:

- **Base:** attivato di default, mostra il messaggio di errore sia sullo standard error che sul documento finale;
- **Strict:** attivato tramite flag `--strict`, stampa lo stack trace su standard error e termina l'esecuzione del programma con l'exit code dato dal codice dell'errore incontrato.

Il seguente diagramma mostra la modellazione delle strategie di error handling appena viste:

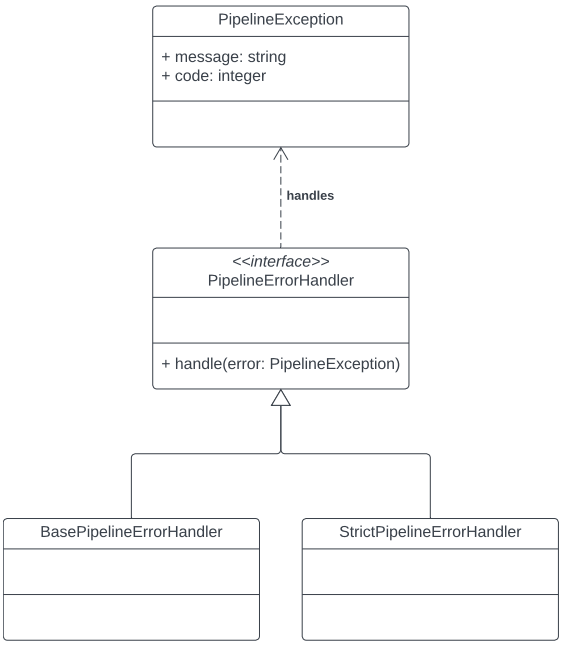


Figura 9.9: Architettura della gestione degli errori.

9.2 Implementazione

Adattamento

Grazie al `KFunctionAdapter`, la seguente funzione Kotlin:

```
fun greet(name: String): StringValue {
    return StringValue("Hello $name!")
}
```

è adattata alla funzione con nome `greet`, parametri `[name: StringValue]` e tipo di output `StringValue`.
È inoltre possibile utilizzare l'annotazione `@Name`, sia sulla funzione che sui parametri, per forzare un nome Quarkdown differente da quello Kotlin. Questo risulta utile in quanto, per scelta, Quarkdown segue la convenzione del `lowercase`, a differenza del `camelCase` di Kotlin. Nel caso dei parametri, definisce il nome a cui riferirsi tramite *named arguments* (il cui funzionamento non verrà approfondito per brevità, ma che continua a basarsi sull'`ArgumentsBinder`).

> **Esempio 20.** Consideriamo la seguente funzione Kotlin:
>
> ```
> @Name("islower")
> fun isLower(a: Int, @Name("than") b: Int): BooleanValue {
>     return BooleanValue(a < b)
> }
> ```
>
> In Quarkdown, tale funzione può essere invocata con:
>
> ```
> .islower {2} than:{5}
> ```
>
> Invocare `.isLower` produce invece errore, in quanto il simbolo non è riconosciuto nel contesto.

Accodamento

Tutti i nodi di tipo `FunctionCallNode` vengono inseriti in una coda all'interno di `Context` durante il parsing, al fine di non dover ripercorrere l'intero albero una seconda volta.

A questo punto, un `FunctionCallNodeExpander` svuota la coda e per ogni nodo esegue la sua corrispondente chiamata a funzione, ottenuta tramite una chiamata a `Context#resolveUnchecked` e quindi di tipo `UncheckedFunctionCall`, che dietro le quinte chiama `getFunctionByName` in modo *lazy* al bisogno, lanciando solo allora un errore nel caso di simbolo non risolto.
L'expander, ottenuto l'output della chiamata, delega a un `Visitor` la conversione di quest'ultimo in un `Node` che possa essere aggiunto ai figli del `FunctionCallNode`, e quindi anche all'AST, come precedentemente accennato, a seconda del tipo di valore restituito.
Svuotata interamente la coda, il processo è concluso, il documento contiene gli output delle funzioni e si può procedere al rendering del documento.

Esportazione delle librerie

Definita la generica `MultiLibraryLoader<S>` e implementato il suo metodo `load(source: Set<S>)` tramite flat-map dei sotto-loaders, l'implementazione di `MultiFunctionLibraryLoader` sulla base di tanti `FunctionLibraryLoader` risulta banale.
È possibile dunque implementare l'esportazione della libreria, in questo caso `stdlib`, a partire da un suo singleton:

```
object Stdlib : LibraryExporter {
    override val library: Library
        get() =
            MultiFunctionLibraryLoader(name = "stdlib").load(
                setOf(
                    ::function1,
                    ::function2,
                    ...
                )
            )
}
```

9.2.1 Funzioni di flusso

Tra le funzioni offerte da `stdlib` spiccano per particolarità quelle *di flusso*, ossia funzioni che aggiungono logica di scripting al documento e che rievocano il concetto di *statement* dei linguaggi di programmazione. A questa categoria appartengono:

- `if`: mostra del contenuto se una condizione è soddisfatta;
- `ifnot`: come `if`, ma inversa;
- `foreach`: mostra del contenuto ripetuto per ogni elemento passato in argomento, che può essere un iterabile o un range di numeri;
- `function`: definisce una funzione che può essere chiamata;
- `var`: definisce o sovrascrive una variabile;
- `let`: crea una variabile temporanea utilizzabile in un singolo scope.

if

La funzione `if` è così definita:

```
@Name("if")
fun `if`(
    condition: Boolean,
    body: Lambda,
): OutputValue<*> =
    when (condition) {
        true -> body.invokeDynamic()
        false -> VoidValue
    }
```

Quest'implementazione contiene alcuni spunti interessanti:

- `context` è un parametro *iniettato* e che quindi non richiede esplicitamente un argomento, bensì è il `Binder` ad associare staticamente il suo valore corrispondente, in questo caso il contesto di esecuzione;
- `body` contiene il contenuto del blocco. Potrebbe essere stato passato come `Node`, ma ciò avrebbe elaborato inutilmente il blocco nel caso la condizione non fosse soddisfatta.
- Non soddisfatta la condizione, viene ritornato un valore vuoto. Si ricordi che tutte le funzioni esportate devono sempre ritornare un'istanza di `OutputValue`, per cui viene ritornato il singleton `VoidValue`.

Esempio 21.

```
.if {.islower {2} than:{5}}
    **Yes**, 2 is less than 5.
```

foreach

La funzione `.foreach` permette di iterare su una collezione di elementi, a cui appartengono anche i *range* `N..M`. L'elemento corrente è salvato nel primo ed unico argomento del lambda, che può essere esplicito o implicito.

Esempio 22. Esplicito:

```
.foreach {1..3}
    n:
    The number is **.n**
```

Implicito:

```
.foreach {1..3}
    The number is **.1**
```

Output:
*The number is **1***
*The number is **2***
*The number is **3***

Come zucchero sintattico, è possibile rendere impliciti gli estremi di un range. Rimuovere l'estremo di sinistra lo farà iniziare da 1, quello di destra lo farà finire a $+\infty$. È inoltre possibile eseguire operazioni con il placeholder.

Esempio 23.

```
.foreach {..3}
    The number is **.multiply {.1} {.1}**
```

> *Output:*
> *The number is **1***
> *The number is **4***
> *The number is **9***

function

Una funzionalità particolare è la possibilità di dichiarare funzioni all'interno di un sorgente tramite la funzione `.function`, la cui signature è:

```
fun function(
    @Injected context: MutableContext,
    name: String,
    body: Lambda,
): VoidValue
```

> **Esempio 24.**
>
> ```
> .function {greet}
>     to from:
>     Hello **.to** from .from!
>
> .greet {Giorgio} {Michele}
> ```
>
> *Output: Hello **Giorgio** from Michele!*

Quello che avviene è:

1. Vengono ottenuti gli argomenti espliciti del lambda `body`, definendo così i parametri della funzione: nell'esempio, chiamare `.greet {Giorgio}` avrebbe causato un errore in quanto sono attesi due argomenti!
2. Viene creato una `SimpleFunction` a cui vengono passati nome della funzione, parametri attesi e funzione `invoke()`. Quello che avviene in `invoke()` è semplicemente una sostituzione testuale dei placeholder con gli argomenti passati, e l'output è l'*evaluation* di questa stringa.
3. La funzione viene registrata nel contesto, che è stato ottenuto grazie al parametro iniettato.

Esempio 25. L'output della funzione è *dinamico*, esattamente come quello che l'utente scrive tra gli argomenti in quanto non si conosce il suo tipo a priori. È quindi assolutamente possibile scrivere ed eseguire questo tipo di codice:

```
.function {mysum}
    a b:
    .sum {.a} {.b}

2*(2+1) = .multiply {2} by:{.mysum {2} {1}}
```

Output: `2*(2+1) = 6`

Esempio 26. Lo scripting di Quarkdown è Turing complete: È possibile scrivere funzioni discretamente complesse, ecco un esempio che calcola l'`N`-esimo numero di Fibonacci con un approccio ricorsivo:

```
.function {fib}
    n:
    .if { .islower {.n} than:{2} }
        .n
    .ifnot { .islower {.n} than:{2} }
        .sum {
            .fib { .subtract {.n} {1} }
        } {
            .fib { .subtract {.n} {2} }
        }

$ F_5 $ = .fib {5}
```

Si noti come non sia presente una keyword di `return`, bensì l'output della funzione è strettamente il contenuto che verrebbe mostrato se fosse eseguito come plain Markdown.

Capitolo 10

Attraversamento dell'albero

10.1 Design

10.1.1 Contesto e attributi

Problema I nodi di tipo `ReferenceLink` e `ReferenceImage` posseggono un *label* facente riferimento al *label* di un nodo `LinkDefinition`, contenente l'URL concreto da utilizzare. Il `LinkDefinition` può trovarsi in qualunque punto dell'albero dei nodi, non necessariamente prima del riferimento. Bisogna dunque modellare una strategia di look-up di un link a partire da un *label* di riferimento.

Soluzione La soluzione adottata consiste nell'accodamento di ogni nodo di tipo `LinkDefinition` in una lista esterna (è necessario l'ordinamento in quanto una stessa label può potenzialmente appartenere a più definizioni distinte, per cui si va a selezionare la prima di esse per convenzione) al momento del suo parsing.
Questo dato è conservato in `AstAttributes`, che mantiene le informazioni legate all'albero ed è implementato da semplici data class, a cui accede `Context`, astrazione dei dati condivisi all'interno della pipeline.
Dal momento che `Context` è aggiornato solamente in certe fasi della pipeline, si distinguono contesti mutabili e non mutabili al fine di aggiungere un livello di protezione dei dati, che vanno esclusivamente letti - e non modificati - nelle altri fasi della pipeline.

L'accodamento di informazioni nel contesto corrente, nella modalità appena descritta, è modellato dal seguente diagramma:

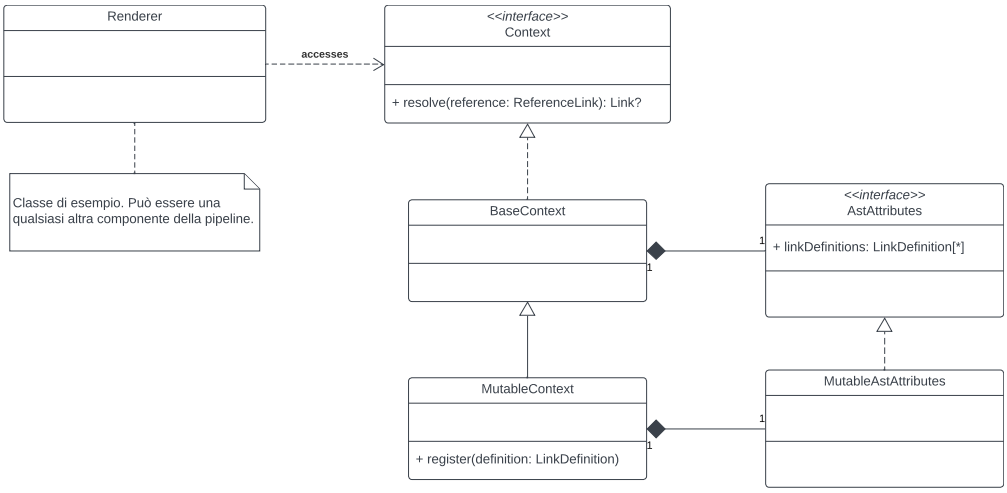


Figura 10.1: Architettura di contesto e attributi.

10.1.2 Aggiornamento degli attributi

Problema Creata la struttura in cui salvare gli attributi, è necessario salvarli in qualche modo. Tornando all'esempio dei `LinkDefinition`, vogliamo trovare e salvare *tutti* i nodi di quel tipo.

Vecchia soluzione In una prima versione del software, questo aggiornamento era effettuato gradualmente in fase di parsing: prima di ritornare un nodo di tipo `LinkDefinition` lo si aggiungeva agli attributi. Ma che succede se un nodo di questo tipo viene ritornato dinamicamente da una funzione, e non elaborato nel parser? È quindi necessario trovare un approccio più sofisticato.

Soluzione Si aggiunge una nuova piccola fase della pipeline in cui viene percorso l'intero albero dei nodi, da radice a foglie. Per poter garantire scalabilità, la visita di un nodo non esegue direttamente operazioni, ma è possibile registare delle azioni, tramite observable pattern, eseguite quando un nodo di un certo tipo è incontrato.

L'attraversamento dell'albero è eseguito da un oggetto di tipo `AstIterator` e, al fine di mantenere lo stesso stile adottato in tutta la pipeline, esso viene generato da una factory contenuta dal flavor.

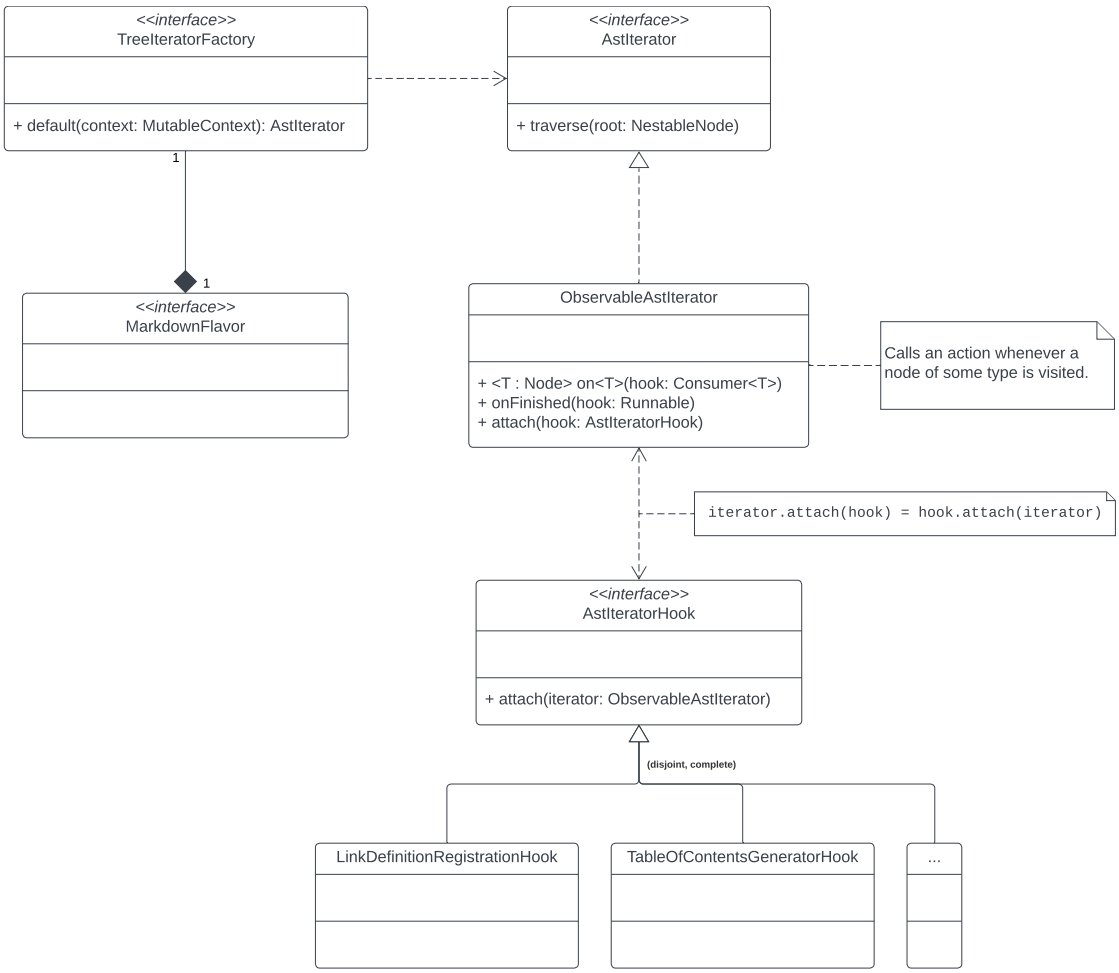


Figura 10.2: Architettura del tree visit.

10.2 Implementazione

Il sistema di visite osservabili è così implementato nella classe `ObservableAstIterator`:

```
/**
 * Hooks that will be called when a node of a certain type is visited.
 */
val hooks: MutableList<(Node) -> Unit> = mutableListOf()

/**
 * Registers a hook that will be called when a node of type [T] is visited.
 * @param hook action to be called, with the visited node as parameter
 * @param T desired node type
 * @return this for concatenation
 */
inline fun <reified T : Node> on(noinline hook: (T) -> Unit) =
    apply {
        hooks.add {
            if (it is T) hook(it)
        }
    }

override fun traverse(root: NestableNode) {
    root.flattenedChildren().forEach { node ->
        hooks.forEach { hook -> hook(node) }
    }
    // Qui sono inoltre invocati gli onFinished
}
```

Dove `NestableNode#flattenedChildren` è una funzione di utility che ritorna una sequenza (o, in Java, una *stream*) di tutti i nodi figli tramite DFS.
Grazie alla combinazione `inline` + `reified`, si raggira la *type erasure* della JVM permettendo un approccio molto elegante:

```
iterator.on<T> { t -> ... }
```

Esempio 27. L'esempio seguente effettua la registrazione dei `LinkDefinition` negli attributi ogni volta che un nodo di quel tipo è incontrato:

```
/**
 * Hook that registers the [LinkDefinition]s in the [context]
 * so that they can be later looked up
 * by [ReferenceLink]s.
 */
class LinkDefinitionRegistrationHook(
    private val context: MutableContext
) : AstIteratorHook {
    override fun attach(iterator: ObservableAstIterator) {
        iterator.on<LinkDefinition> {
            context.register(it)
        }
    }
}
// ...
iterator
    .attach(LinkDefinitionRegistrationHook(context)
    .traverse(root)
```

Esempio 28. L'esempio seguente tiene traccia di tutti i titoli presenti, per poi delegare ad un componente esterno la generazione dell'indice di un documento:

```
/**
 * Hook that allows the generation of a [TableOfContents]
 * by iterating through [Heading]s.
 * The [TableOfContents] is stored in the [context]'s
 * [MutableContext.attributes] at the end of the traversal.
 */
class TableOfContentsGeneratorHook(
    private val context: MutableContext
) : AstIteratorHook {
    override fun attach(iterator: ObservableAstIterator) {
        val headings = iterator.collectAll<Heading>()

        iterator.onFinished {
            context.attributes.tableOfContents =
                TableOfContents.generate(headings.asSequence())
        }
    }
}
```

con `collectAll` metodo di utility:

```
/**
 * Collects all the visited nodes of type [T]
 * into a collection.
 * @param T node type
 * @return an ordered list (DFS order)
 * containing all the visited nodes
 * of type [T] in the tree
 */
inline fun <reified T : Node> collectAll(): List<T> =
    mutableListOf<T>().apply {
        on<T>(::add)
    }
```

Capitolo 11

Rendering

Il prossimo step della pipeline consiste nella trasformazione dell'albero dei nodi in un output visualizzabile in modo chiaro e non ambiguo dall'utente. Come per tutti i compilatori per Markdown, l'implementazione standard del renderer produce codice HTML valido e pronto per essere caricato in un browser. In futuro potrebbero venire implementati nuovi linguaggi target - LaTeX per esempio - per cui l'architettura è stata progettata in modo da garantire estendibilità e riusabilità.

11.1 Design

> **Esempio 29.** Si considerino i seguenti esempi di rendering da Markdown a HTML:
>
> - Enfasi:
>
>   ```
>   **testo**
>   ```
>
>   ```
>   testo
>   ```
>
> - Immagini:
>
>   ```
>   ![label](/url.png "Titolo")
>   ```
>
>   ```
>   <img src="/url.png" alt="label" title="Titolo" />
>   ```

- Liste:

```
- A
- B
- C
```

```
<ul>
    <li>A</li>
    <li>B</li>
    <li>C</li>
</ul>
```

- Tabelle:

```
|  A  |  B  |
| --- | --- |
|  C  |  D  |
```

```
<table>
    <thead>
        <tr>
            <th>A</th>
            <th>B</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>C</td>
            <td>D</td>
        </tr>
    </tbody>
</table>
```

Il concetto di base è analogo a quello del parsing: un visitor che, invece che i token, visita i vari tipi di `Node` che possono essere presenti nell'albero.

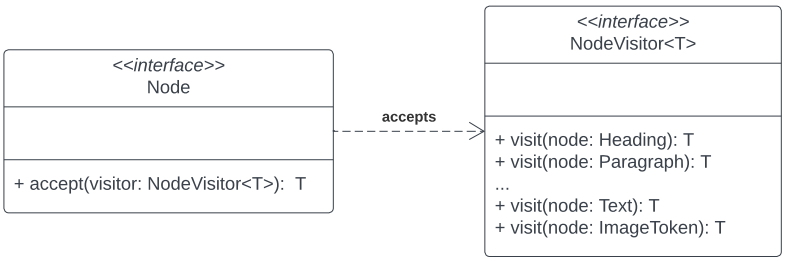


Figura 11.1: Il node visitor

11.1.1 Tag builder

Problema L'approccio che potrebbe venire naturale adottare sarebbe quello di implementare i metodi del visitor restituendo codice HTML scritto 'a mano' come in questo caso:

```
override fun visit(node: CodeSpan) =
    "<code>${node.content}</code>"
```

Seppur non sia un approccio sbagliato, non è complicato individuare i suoi difetti, in primis la difficoltà di lettura (codice e contenuto sono tutt'uno) e di manutenzione (bisogna assicurarsi che le tag di apertura siano sempre chiuse correttamente). Esistono inoltre casi più complessi, come con le immagini, che presentano attributi opzionali, che renderebbero il codice pieno di flussi di controllo. È necessario dunque trovare una soluzione che raffini in modo più elegante il problema.

Soluzione La soluzione converge al *builder* pattern:

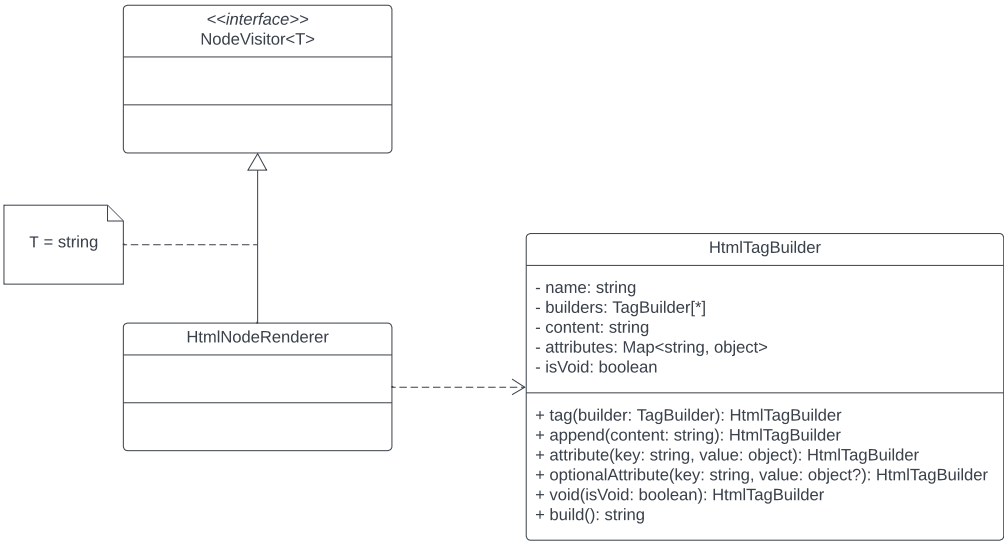


Figura 11.2: Primo prototipo di tag builder HTML

Problema Come menzionato precedentemente, l'architettura deve essere abbastanza flessibile da poter avere implementazioni del renderer per diversi linguaggi di output. Supponendo si voglia aggiungere il supporto alla traspilazione in codice LaTeX, bisognerebbe creare un nuovo builder da zero, che però ha elementi in comune con l'`HtmlTagBuilder`: anche LaTeX dispone di tag (`\nome` è il nome del tag) ed il suo contenuto si trova all'interno di parentesi graffe ma, ad esempio, non ha attributi. È quindi necessario aggiungere un ulteriore livello di astrazione.

Soluzione `NodeRenderer` e `TagBuilder` si pongono da intermediari tra il visitor e le implementazioni concrete per il linguaggio target. Il primo fornisce un factory method che permette la creazione di un `TagBuilder` per il target desiderato.

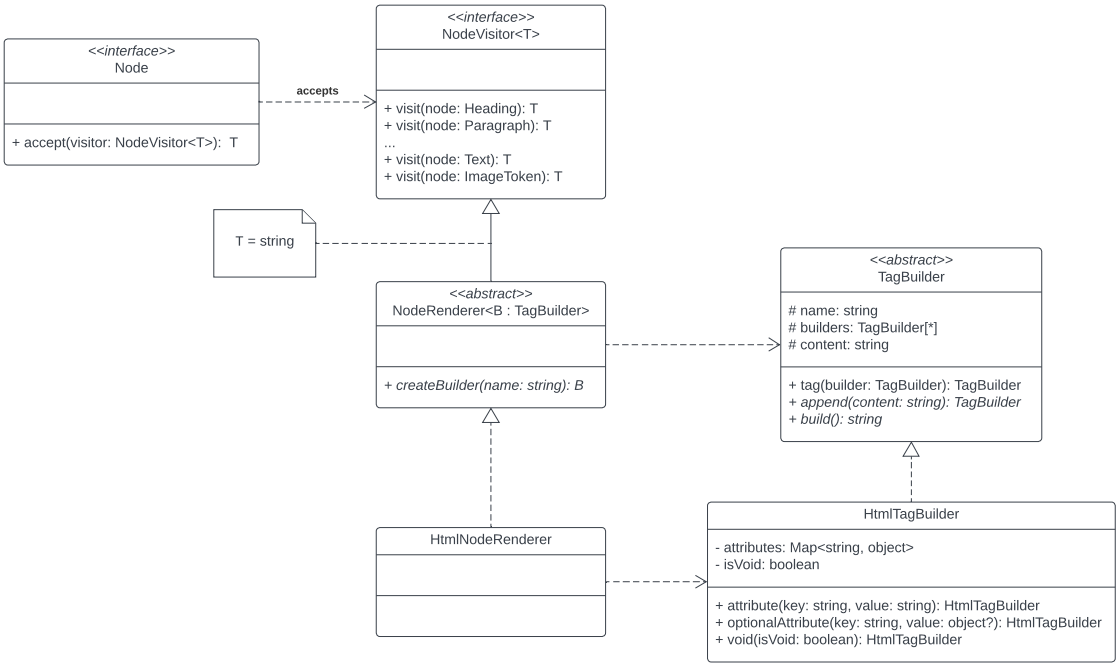


Figura 11.3: Architettura del renderer

11.1.2 Flavor

Per garantire l'estendibilita' servita dai flavor, ognuno di essi mette a disposizione la sua propria specializzazione del renderer attraverso la sua `RendererFactory`:

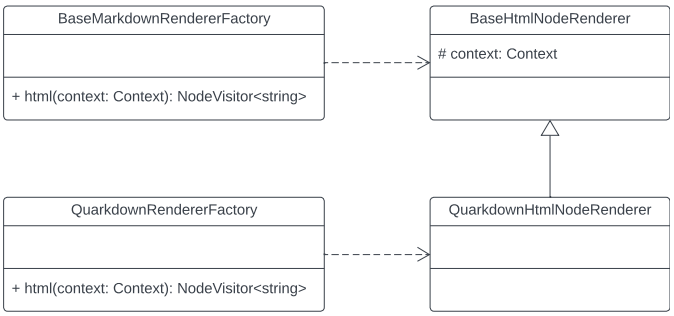


Figura 11.4: Legame tra renderer e flavor.

11.1.3 Architettura finale

Di seguito si mostrano tutti gli elementi che compongono il processo di rendering dei nodi nelle modalità descritte in questo capitolo.

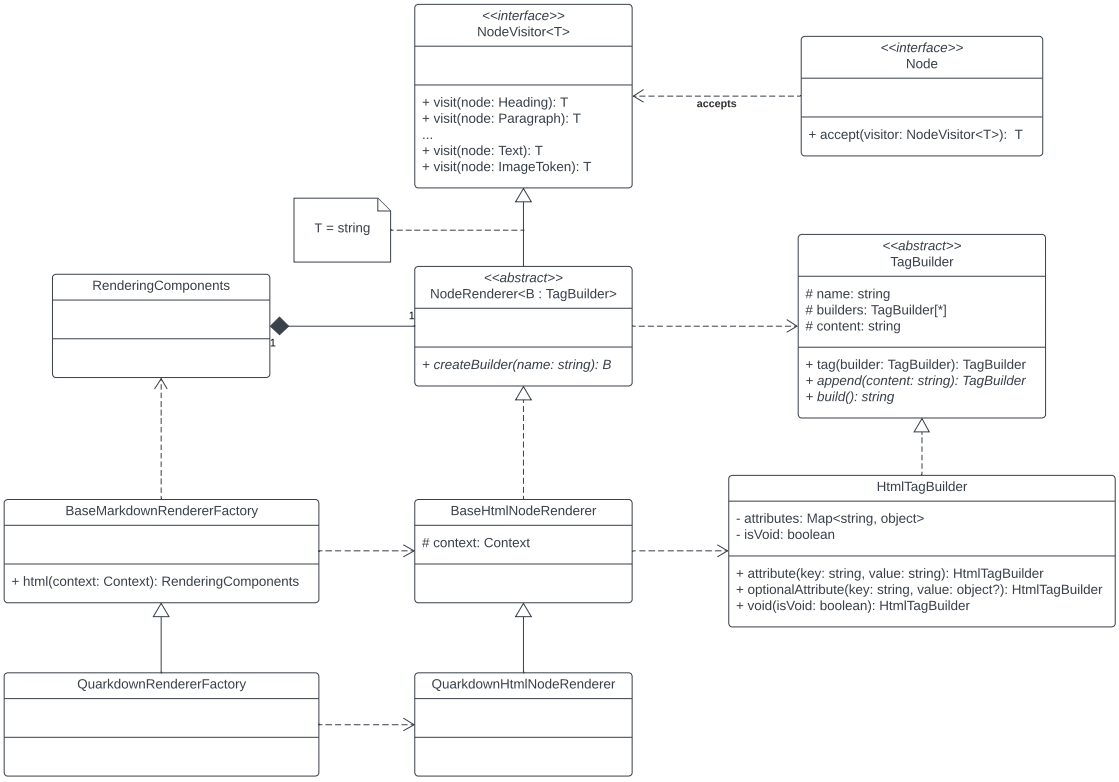


Figura 11.5: Architettura finale del renderer.

11.2 Implementazione

Grazie ai tag builder si ottiene un risultato più leggibile rispetto a quello precedente:

```
class HtmlNodeRenderer : NodeVisitor<CharSequence> {
    ...
    override fun visit(node: CodeSpan) =
        HtmlTagBuilder("code")
            .append(node.content)
            .build()
    ...
}
```

Infine viene aggiunto dello zucchero sintattico avvalendosi dei *builder blocks* di Kotlin creando un vero e proprio domain-specific language (DSL):

```
// Builder di primo livello
fun <B : TagBuilder> NodeRenderer<B>.tagBuilder(
    name: String,
    init: B.() -> Unit,
) = createBuilder(name).also(init)

// Builder innestati
fun TagBuilder.tag(
    name: String,
    init: TagBuilder.() -> Unit,
) = renderer.tagBuilder(name, init).also { this.builders += it }
```

e degli operator overloads dell'operatore `+` su `TagBuilder`:

```
/**
 * Appends a string value to this tag's content.
 * Usage: `+"Some string"`
 */
operator fun CharSequence.unaryPlus() {
    append(this)
}

/**
```

```
 * Appends a node to this tag's content.
 * Their string representation is given by this [TagBuilder]'s [renderer].
 * Usage: `+someNode`
 */
operator fun Node.unaryPlus() {
    +this.accept(renderer)
}

/**
 * Appends a sequence of nodes to this tag's content.
 * Their string representation is given by this [TagBuilder]'s [renderer].
 * Usage: `+someNode.children`
 */
operator fun List<Node>.unaryPlus() {
    forEach { +it }
}
```

Esempio 30. Si consideri il frammento di Markdown `***Testo***`. Questo corrisponde ad un nodo di tipo `StrongEmphasis` con un nodo figlio `Text` e va renderizzato - secondo lo standard - come `<em><strong>Testo</strong></em>`. È possibile implementarlo nel seguente modo rendendo la lettura molto più naturale:

```
...
override fun visit(node: StrongEmphasis) =
    tagBuilder("em") {
        tag("strong") {
            +node.children
        }
    }
    .build()
...
```

Un altro esempio in cui il nuovo builder pattern risulta utile è quello delle immagini:

```
...
override fun visit(node: Image) =
    tagBuilder("img")
        .attribute("src", node.link.url)
        // Si considera solo il contenuto testuale
        // (linea guida CommonMark 6.4)
        .attribute("alt", node.link.label.toPlainText())
        // Il titolo viene aggiunto solo se title non è nullo
        .optionalAttribute("title", node.link.title)
        // Senza tag di chiusura
        .void(true)
        .build()
...
```

Finalmente, il punto di ingresso del renderer si trova nella visita alla radice dell'albero:

```
override fun visit(node: AstRoot) =
    buildTag("html") {
        tag("head") {
            tag("meta")
                .attribute("charset", "UTF-8")
                .void(true)
        }
        tag("body") {
            +node.children
        }
    }
```

L'append dei sotto-nodi tramite l'operator overload di `+` chiama `accept` per ognuno di essi, che quindi torna a chiamare un metodo `visit` del visitor corrente, ripetendo la procedura finchè tutti i nodi visitati sono nodi foglia, eseguendo dunque una visita depth-first e renderizzando l'albero nella sua interezza.

Contesto

Come anticipato, i nodi `ReferenceLink` fanno riferimento ad un link simbolico definito da un `LinkDefinition`, ottenuto a seguito di un look-up via `Context`. Il renderer deve quindi avere un riferimento al contesto per poter rappresentare un riferimento:

```
class BaseHtmlNodeRenderer(private val context: Context)
    : NodeRenderer<HtmlTagBuilder>() {
    ...
    // Si renderizza il nodo di fallback
    // se la reference non è definita
    override fun visit(node: ReferenceLink) =
        (context.resolve(node) ?: node.fallback())
            .accept(this)
    ...
}
```

Con `fallback` un supplier del nodo da utilizzare in caso il lookup fallisca (solitamente un semplice testo).

Capitolo 12

Post rendering

L'ultimo passo della pipeline riguarda il *post rendering*: il risultato del rendering viene in questa fase iniettato all'interno di un template. Nel caso specifico di HTML, l'output della fase precedente viene inserito nel `body` del documento, e sempre questa fase sarà responsabile del collegamento dei temi esterni e di possibili script.
Quest'operazione di inserimento del contenuto in un template è stata chiamata ***render wrapping***.

12.1 Design

12.1.1 Render wrapper

Si pone l'obiettivo di definire un linguaggio di template per l'output finale della pipeline. Nello specifico, un file HTML prodotto dovrà essere valido, dunque dotato di `head`, con metadati, e `body`.
In esso si trovano due elementi:

- Valori: `[[abc]]` verrà sostituito con il valore associato alla chiave `abc`;
- Condizioni: `[[if:abc]] ... [[endif:abc]]` verrà mostrato se il valore booleano associato alla chiave `abc` è vero, altrimenti viene eliminato.

A questo scopo è stata realizzata la classe builder-like `RenderWrapper`, che prende in input il codice template, valori e condizioni per riuscire a restituire il codice finale senza elementi di template tramite il metodo `wrap()`.

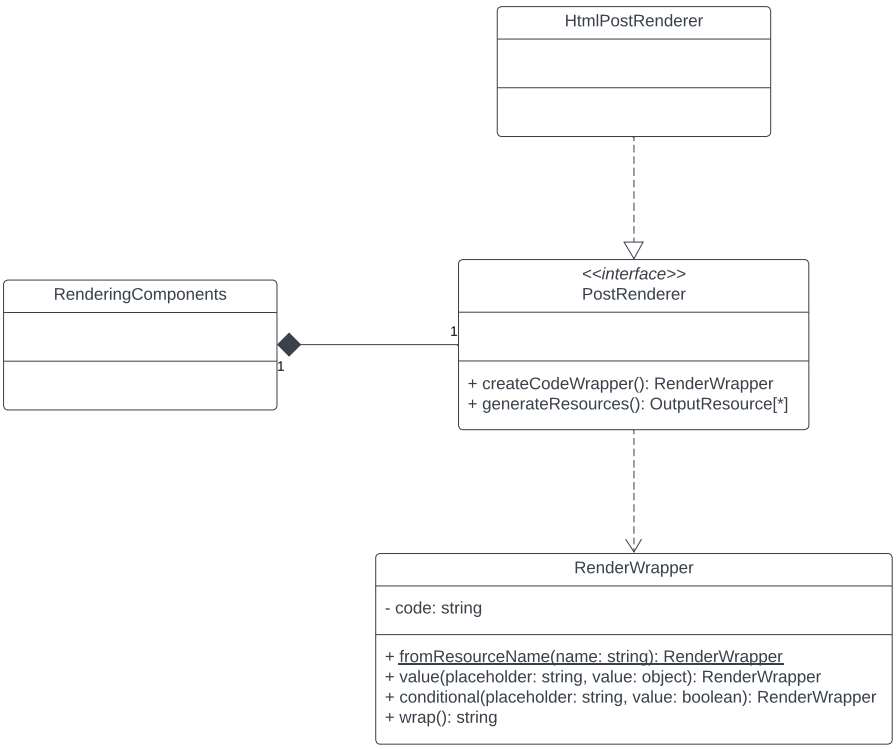


Figura 12.1: Architettura del post renderer.

12.1.2 Tipo di documento

Allo stato attuale, Quarkdown supporta il rendering su tre tipi di documento:

- **Plain:** documento standard, con il contenuto del rendering mostrato senza alcuna impaginazione ad hoc - attivo di default;
- **Paged:** documento diviso in pagine, come un libro. In HTML, l'impaginazione è gestita dalla libreria paged.js;
- **Slides:** documento composto da diapositive, con animazioni ed effetti configurabili. In HTML, ciò è gestito dal framework reveal.js.

Il tipo di documento è impostabile direttamente dal codice Quarkdown tramite chiamata alla funzione `.doctype {plain|paged|slides}`, che aggiorna le informazioni del documento legato al contesto attuale.
Il layout corrispondente alla tipologia di documento viene caricato grazie alle condizioni supportate dal linguaggio di template nel render wrapper:

```
<head>
  ...
  [[if:PAGED]]
  <script src="paged.polyfill.js"></script>
  [[endif:PAGED]]
  [[if:SLIDES]]
  <script src="reveal.js"></script>
  <link rel="stylesheet" href="reveal.css">
  [[endif:SLIDES]]
  ...
</head>
```

12.1.3 Dettagli del documento

Come per `.doctype`, funzioni come `.docname`, `.docauthor`, `.theme` e `.pageformat` permettono di modificare i dettagli del documento finale. Questi dati sono salvati nella classe mutabile `DocumentInfo` associata univocamente ad un contesto di esecuzione, come mostrato nel seguente diagramma:

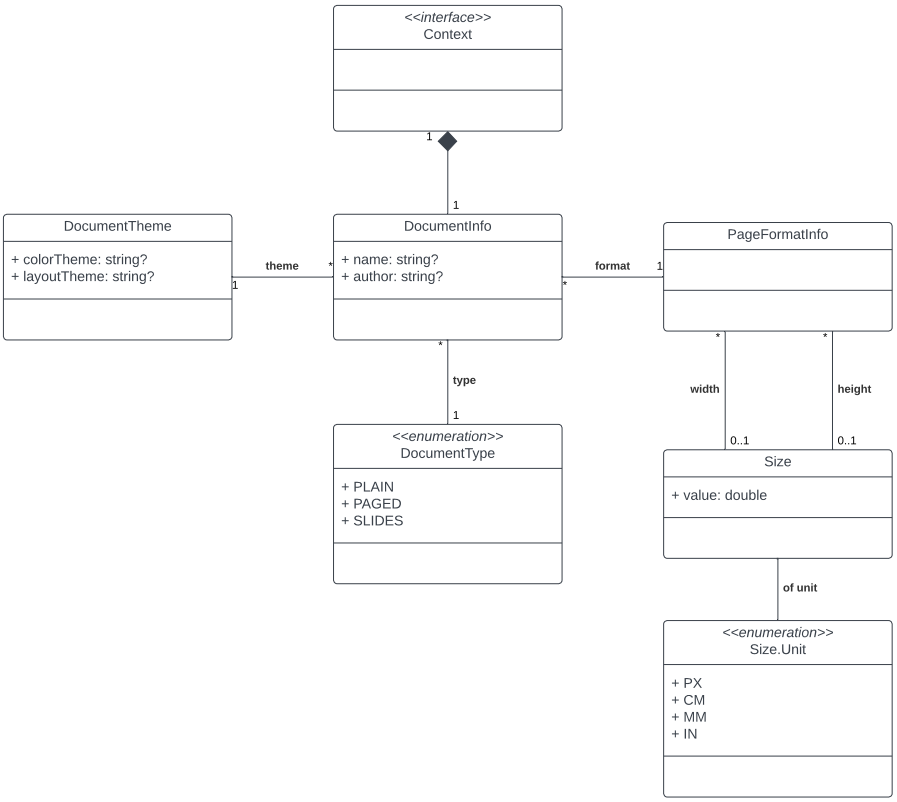


Figura 12.2: Architettura dei dettagli del documento.

12.1.4 Risorse

Il documento di output potrebbe far riferimento a file locali a cui fare riferimento, come un tema o uno script.
Al fine di tracciare di queste risorse, il metodo `PostRenderer#generateResources` ritorna l'insieme delle risorse da esportare. Quest'argomento verrà trattato in modo più completo in sottosezione 13.1.1.

12.2 Implementazione

Creato un `RenderWrapper`, o caricato da `InputStream` tramite il metodo statico `RenderWrapper.fromResourceName`, è possibile usare la sua interfaccia builder-like e chiamare i metodi `value(placeholder: String, value: Any)` e `conditional(placeholder: String, value: Boolean)`, che salvano tali dati in mappe, rispettivamente `values` e `conditionals`.
Chiamato il build method `wrap()`, il codice 'di template' viene manipolato:

- Per ogni `conditional` vengono cercati i blocchi di tipo `[[if:placeholder]]...[[endif:placeholder]]` tramite regex. Se la condizione corrispondente è vera si eliminano i delimitatori, altrimenti si elimina l'intero blocco.
- Per ogni `value` vengono rimpiazzati i `[[placeholder]]` con il contenuto testuale (`toString()`) del valore corrispondente.

Capitolo 13

Assemblaggio della pipeline

13.1 Design

Creati gli elementi cardine della pipeline - lexer, parser, function expander, tree iterator, renderer e post renderer - ciò che rimane è collegarli tra loro creando un flusso sequenziale di input e output tra una fase e l'altra.

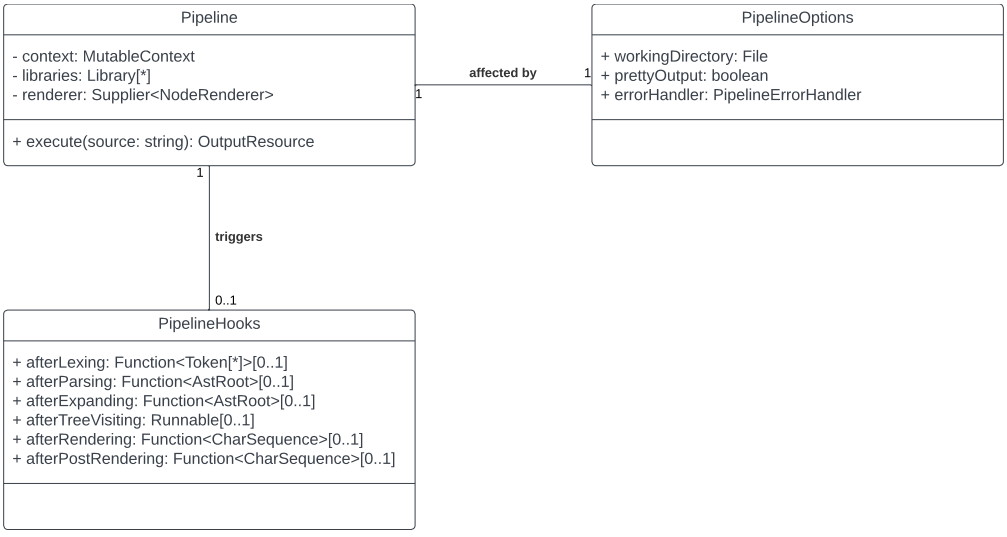


Figura 13.1: Architettura della pipeline.

L'utilizzo degli hooks consente l'adozione dell'*observer* pattern, rendendo possibile l'esecuzione di azioni al termine di ogni fase della pipeline, anche banalmente per stampare log di debug. Le opzioni, definite dalle flag a riga di comando, consentono l'alterazione di specifici aspetti della pipeline.

13.1.1 Esportazione su file

Problema La chiamata a `execute` richiama gli `hooks` ad ogni passaggio della pipeline. Tuttavia, si vorrebbe supportare l'esportazione del documento finale su file. Si consideri che, soprattutto con il rendering in HTML, il prodotto finale sarà composto da più file (index HTML, tema CSS, e potenzialmente script JavaScript).

Soluzione `execute` restituisce una *risorsa*, che astrae ad alto livello un file, senza però fare esplicitamente riferimento ad esso. Una risorsa può essere un artefatto (rappresentato da un file testuale (`TextOutputArtifact`) o binario (`BinaryOutputArtifact`)) o un gruppo di altre sotto-risorse (rappresentato da una directory). Addizionalmente, un artefatto può essere *lazy* se il suo contenuto va caricato solo al bisogno - questo viene utilizzato nel caricamento delle risorse da `InputStream`, come immagini o temi.

Viene infine utilizzato un visitor, pattern soventemente adottato in questo progetto, per permettere l'operazione di salvataggio su file di una risorsa, mantenendo così il livello di astrazione che rende le risorse agnostiche alla loro rappresentazione sul disco.

Le componenti che gestiscono la manipolazione delle risorse nelle modalità precedentemente descritte sono modellate dal seguente diagramma:

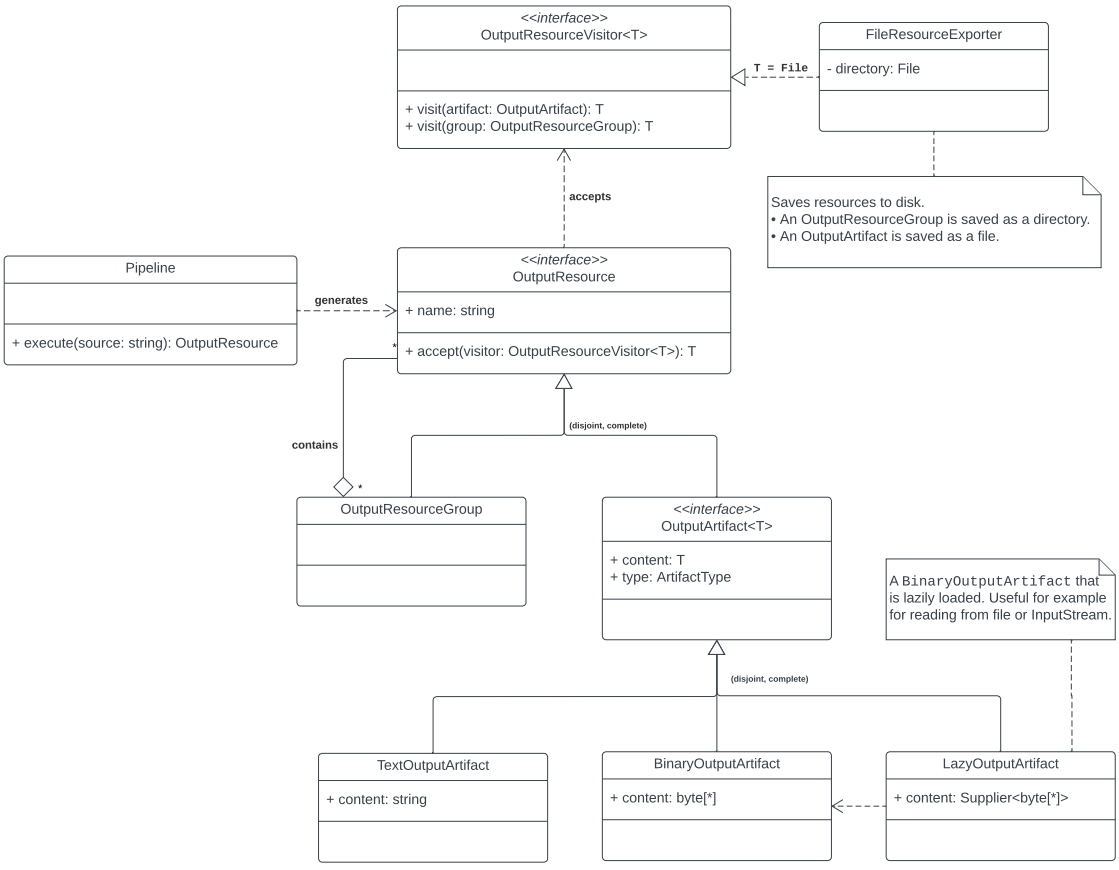


Figura 13.2: Architettura delle risorse.

13.2 Implementazione

Un esempio di flusso di inizializzazione ed esecuzione è il seguente:

```
// Flavor da utilizzare in tutta la pipeline
val flavor: MarkdownFlavor = QuarkdownFlavor

// Azioni da eseguire dopo ogni fase
val hooks =
    PipelineHooks(
        afterLexing = { tokens ->
            ...
        },
        afterParsing = { document ->
            ...
        },
        afterRendering = { rendered ->
            ...
        },
        afterPostRendering = { wrapped ->
            ...
        },
    )

// Inizializzazione della pipeline
val pipeline =
    Pipeline(
        context = MutableContext(flavor),
        libraries = setOf(...),
        renderer =
            { rendererFactory, context ->
                rendererFactory.html(context)
            },
        hooks,
    )

// Esecuzione
pipeline.execute(source)
```

Ad esempio, nel main del modulo `cli` si inizializzano gli hooks che si occupano di

effettuare log dei risultati intermedi in *debug* mode, e di quello finale in *info* mode.

Per quanto riguarda le risorse, `execute` ottiene l'insieme di risorse dal post-renderer (che quindi, come accennato, nel caso di HTML saranno index, tema e possibilmente script) e le inserisce in un `OutputResourceGroup` che viene infine ritornato. Il salvataggio del gruppo di risorse su file, in una directory, avviene banalmente:

```
...
val resource = pipeline.execute(source)
resource.accept(FileResourceExporter(directory))
```

Capitolo 14

Dimostrazione

14.1 Strumenti a confronto

In questa sezione verrà confrontato Quarkdown con i suoi principali competitor analizzati in sezione 2.1, analizzandone pro, contro e casi d'uso, fornendo vari esempi che coprono aree diverse del linguaggio.

14.1.1 Quarkdown vs. LaTeX

	LaTeX	Quarkdown
Semplice e leggibile	No	Sì
Controllo del layout	Sì	Sì
Scripting	Parziale	Sì
Internazionalizzazione	Sì	No (pianificata)
Generazione dell'indice	Sì	Sì
Bibliografia `.bib`	Sì	No
Canvas	Sì	No (pianificata)
Target	PDF	HTML (+ PDF via browser)
Tipo di documento	Articoli, libri, presentazioni	Libri, articoli, presentazioni, inline

Seppur più completo e con una grande quantità di package creati dalla community, LaTeX risulta molto più verboso di Quarkdown, richiedendo spesso più codice del dovuto per ottenere semplici impostazioni del layout.

LaTeX

```latex
\tableofcontents

\section{Section}

\subsection{Subsection}

\begin{enumerate}
    \item \textbf{First} item
    \item \textbf{Second} item
\end{itemize}

\begin{center}
    This text is \textit{centered}.
\end{center}

\begin{figure}[!h]
    \centering
    \begin{subfigure}[b]
        \includegraphics[width=0.3\linewidth]{img1.png}
    \end{subfigure}
    \begin{subfigure}[b]
        \includegraphics[width=0.3\linewidth]{img2.png}
    \end{subfigure}
    \begin{subfigure}[b]
        \includegraphics[width=0.3\linewidth]{img3.png}
    \end{subfigure}
\end{figure}
```

Quarkdown

```quarkdown
.tableofcontents

# Section

## Subsection

1. **First** item
2. **Second** item
```

```
.center
    This text is _centered_ .

.row alignment:{spacebetween}
    ![Image 1](img1.png)

    ![Image 2](img2.png)

    ![Image 3](img3.png)
```

14.1.2 Quarkdown vs. AsciiDoc

	AsciiDoc	Quarkdown
Semplice e leggibile	Sì	Sì
Controllo del layout	No	Sì
Scripting	No	Sì
Sintassi	Proprietaria	Markdown + estensione
Formule matematiche	No	Sì
Footnotes	Sì	No
Target	HTML	HTML (+ PDF via browser)
Tipo di documento	Libri, articoli, presentazioni, docs, inline	Libri, articoli, presentazioni, inline

AsciiDoc

```
= Section

A paragraph with *bold* and _italic_ text.

.Image title
image::image.jpg[Alt, align=center]

This is a https://github.com[link].

[quote]
This is a quote.

IMPORTANT: This is a callout.

include::chapter-1.adoc[leveloffset=1]
```

Quarkdown

```
# Section

A paragraph with **bold** and _italic_ text.

![Alt](image.jpg "Image title")
```

```
This is a [link](https://github.com).

> This is a quote.

.box
    This is a callout.

.include {chapter-1.qmd}
```

14.1.3 Quarkdown vs. Quarto

	Quarto	Quarkdown
Controllo del layout	No	Sì
Scripting	No	Sì
Sintassi	Markdown	Markdown + estensione
Integrazione Jupyter	Sì	No
Target	PDF, HTML, DOCX	HTML (+ PDF via browser)
Tipo di documento	Libri, articoli, presentazioni	Libri, articoli, presentazioni, inline

Quarto consente l'impostazione di metadati e altre informazioni tramite un particolare header, mentre questi aspetti in Quarkdown possono essere impostati tramite funzioni.

Quarto

```
---
title: "Doc title"
author: "Author"
toc: true
format:
  pdf:
    geometry:
      - top=30mm
      - left=20mm
---
```

Quarkdown

```
.doctitle {Doc title}
.docauthor {Docauthor}
.pageformat margin:{30mm 20mm}
.tableofcontents
```

14.1.4 Quarkdown vs. Typst

	Typst	Quarkdown
Semplice e leggibile	Sì	Sì
Controllo del layout	Sì (più completo)	Sì
Scripting	Sì	Sì
Sintassi	Proprietaria	Markdown + estensione
Internazionalizzazione	Sì	No (pianificata)
Canvas	Sì	No (pianificata)
Target	PDF	HTML (+ PDF via browser)
Tipo di documento	Articoli, libri, presentazioni	Libri, articoli, presentazioni, inline

Typst risulta un avversario molto più maturo di Quarkdown, con innumerevoli pacchetti disponibili e con una versatile sintassi proprietaria, seppur differente il tipo di target dei due strumenti.

Typst

```
#set page(
  paper: "a4",
  numbering: "1",
)

= Title

#grid(
  columns: (1fr, 1fr),
  align(center)[
    *Therese Tungsten* \
    Artos Institute \
    #link("mailto:tungsten@example.com")
  ],
  align(center)[
    *Dr. John Doe* \
    Artos Institute \
    #link("mailto:doe@example.com")
  ]
)
```

```
#lorem(10)
```

Quarkdown L'attributo `numbering` non è supportato, ma è possibile scegliere un *tema di layout* che lo utilizzi.

```
.pageformat {A4}
.theme layout:{mytheme}

# Title

.grid columns:{2} alignment:{spacearound}
    .center
        **Therese Tungsten**
        Artos Insitute
        [tungsten@example.com](mailto:tungsten@example.com)

    .center
        **John Doe**
        Artos Insitute
        [doe@example.com](mailto:doe@example.com)

.loremipsum
```

Quarkdown vs. MDX

	MDX	Quarkdown
Controllo del layout	No	Sì
Scripting	Sì	Sì
Sintassi	Markdown + HTML	Markdown + estensione
Target	HTML	HTML (+ PDF via browser)
Tipo di documento	Inline	Libri, articoli, presentazioni, inline

Nell'esempio seguente si assume che `MyChart` e `.mychart` siano, rispettivamente, un componente MDX ed una funzione Quarkdown definite dall'utente in un file `snowfall` separato.

MDX

```
import {MyChart} from './snowfall.js'
export const year = 2013

# Last year's snowfall

In {year}, the snowfall was above average.
It was followed by a warm spring which caused
flood conditions in many of the nearby rivers.

<MyChart year={year} />
```

Quarkdown

```
.include {snowfall.qmd}
.var {year} {2013}

# Last year's snowfall

In .year, the snowfall was above average.
It was followed by a warm spring which caused
flood conditions in many of the nearby rivers.

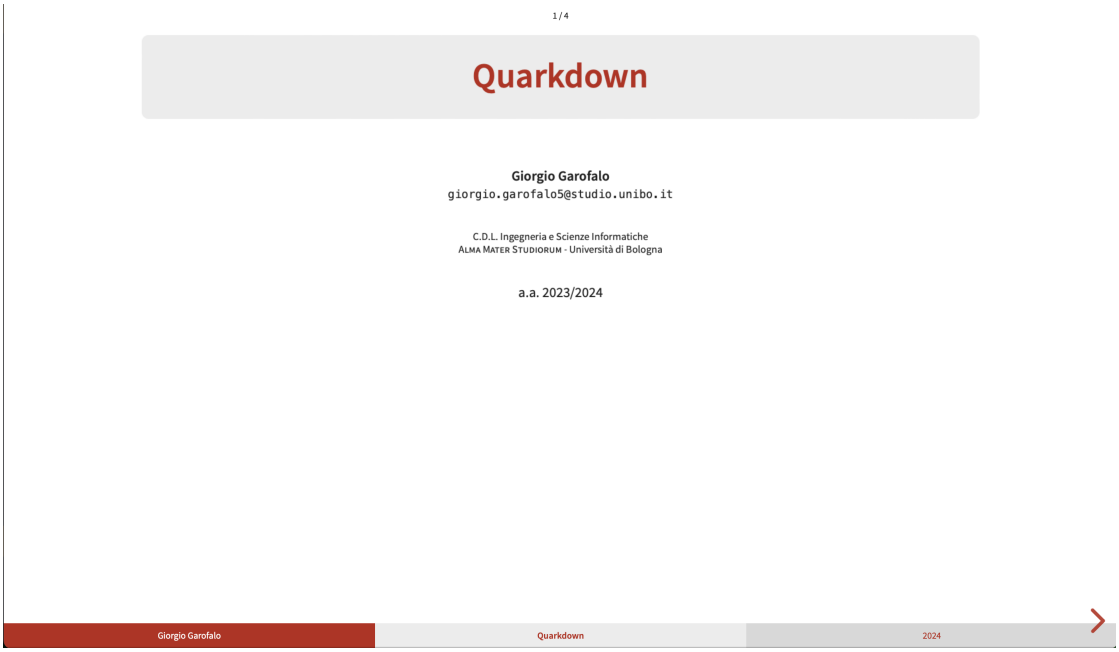
.mychart {.year}
```

14.2 Presentazione di esempio

Le coppie immagine-codice che seguono rappresentano output (documento finale, mostrato sul browser e privo di modifiche) e input (codice Quarkdown).

Oltre agli esempi qui proposti, una dimostrazione interattiva è disponibile e accessibile dalla repository del progetto.

Disclaimer *Data l'attuale immaturità del progetto, future versioni del linguaggio potrebbero subire variazioni rispetto agli esempi qui proposti.*




```
.docname {Quarkdown}
.docauthor {Giorgio Garofalo}
.doctype {slides}
.theme {beaver} layout:{beamer}

.footer
    .docauthor

    **.docname**

    2024

.pagemargin {topcenter}
    .currentpage / .totalpages

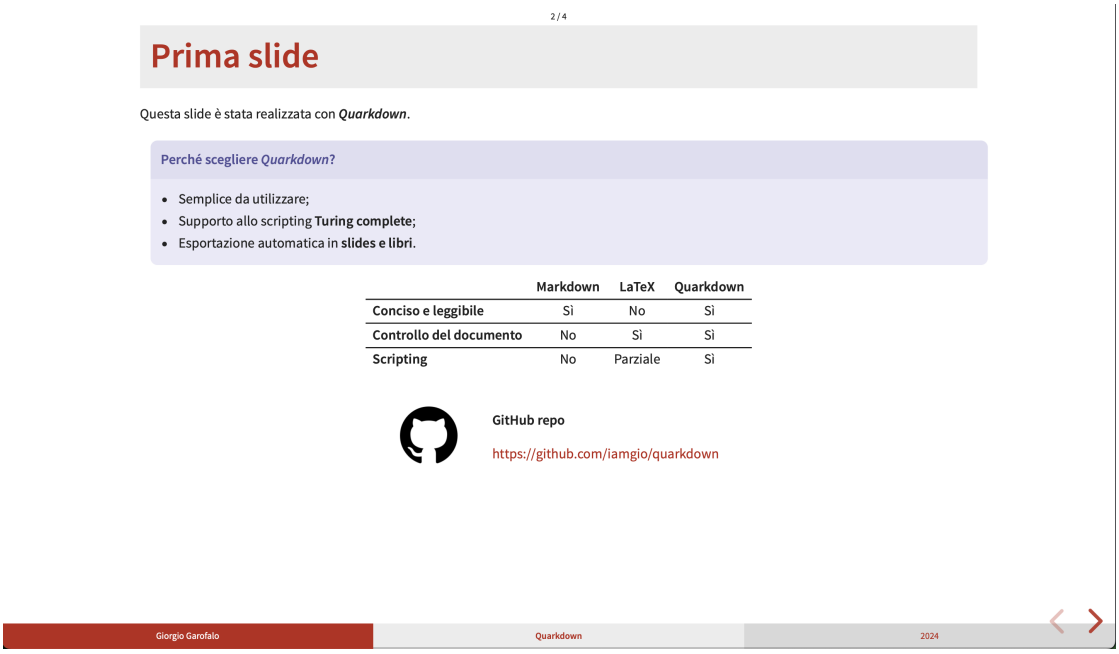
.center
    .box
        # .docname
```

```
.column gap:{12px}
    .whitespace height:{12px}

    **.docauthor**
    `giorgio.garofalo5@studio.unibo.it`

    .text size:{small}
        C.D.L. Ingegneria e Scienze Informatiche
        .text variant:{smallcaps} content:{Alma Mater Studiorum -}
        Università di Bologna

    a.a. 2023/2024
```




```
# Prima slide
Questa slide è stata realizzata con ***Quarkdown***.

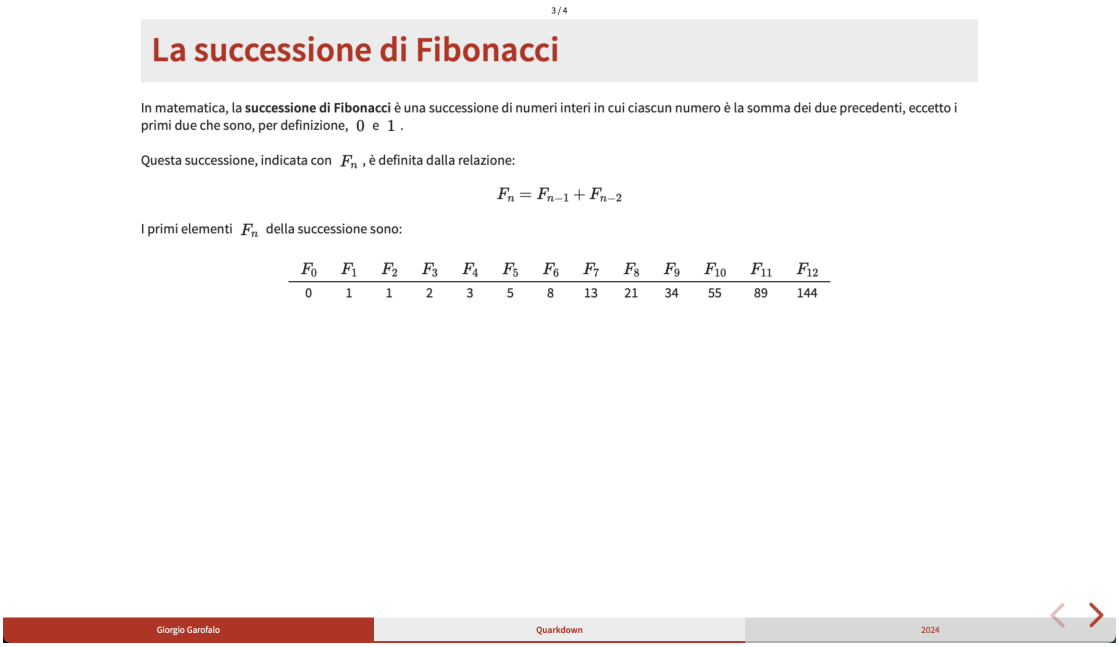
.box {Perché scegliere *Quarkdown*?}
    - Semplice da utilizzare;
    - Supporto allo scripting **Turing complete**;
    - Esportazione automatica in **slides e libri**.

|                            | Markdown |  LaTeX   | Quarkdown |
|----------------------------|:--------:|:--------:|:---------:|
| **Conciso e leggibile**    |    Sì    |    No    |    Sì     |
| **Controllo del documento** |    No    |    Sì    |    Sì     |
| **Scripting**              |    No    | Parziale |    Sì     |

.center
    .row cross:{center} gap:{1cm}
        ![](https://cdn-icons-png.flaticon.com/512/25/25231.png)

        .column cross:{start}
```

```
#### GitHub repo
https://github.com/iamgio/quarkdown
```




```
...

# La successione di Fibonacci

In matematica, la **successione di Fibonacci** è
una successione di numeri interi in cui ciascun numero
è la somma dei due precedenti, eccetto i primi due
che sono, per definizione, $ 0 $ e $ 1 $.

Questa successione, indicata con $ F_n $, è definita dalla relazione:

$ F_n = F_{n-1} + F_{n-2} $

I primi elementi $ F_n $ della successione sono:

.var {t1} {0}
.var {t2} {1}

.table
```

```
.foreach {0..12}
    n:
    | $ F_{.n} $ |
    |:----------:|
    |    .t1     |
    .var {tmp} {.sum {.t1} {.t2}}
    .var {t1} {.t2}
    .var {t2} {.tmp}
```


```
...

# Un po' di codice
La classe `Point` si mostra nel seguente modo:

.code {java}
    .read {Point.java}

Soffermiamoci sul suo _costruttore_ :

.code {java} linenumbers:{no}
    .read {Point.java} lines:{5..8}
```


```
...

# Layout particolari

.row alignment:{spacearound} cross:{start} gap:{2cm}
    .column cross:{start}
        *Quarkdown* permette la realizzazione di layout complessi
        tramite un semplice **approccio imperativo**.

        .foreach {..2}
            .whitespace height:{1cm}
            #### Sezione .1
            .loremipsum

    .column gap:{5mm}
        .foreach {..2}
            .clip {circle}
                ![](img1.jpg)
```

```
.clip {circle}
    ![](img2.jpg)
```

Capitolo 15

Conclusioni

Ricapitolando, Quarkdown è stato ideato e progettato con l'obiettivo di essere un linguaggio di markup versatile, semplice e allo stesso tempo completo, che permetta la realizzazione di documenti di vario tipo più o meno complessi.

Analizzando i diversi snippet di codice mostrati nel precedente capitolo, risulta lo strumento la cui lettura e scrittura del codice risulta **più naturale** agli occhi di chi ha un'esperienza pregressa con Markdown, quasi come se fosse linguaggio naturale.
In quanto a completezza, è in grado di ricreare la quasi totalità degli esempi realizzabili con i competitor, ad eccezione delle funzionalità più avanzate di LaTeX e Typst (in quei casi sarà necessario adottare un tema preimpostato che soddisfi le esigenze). Tra i competitor proposti, ritengo che solamente Typst risulti superiore in termini di controllo ed esperienza utente.

Si può dunque affermare che l'obiettivo primario, il trade-off tra semplicità e completezza, sia stato ampiamente ottenuto. È interessante vedere come poche righe di codice riescano a produrre un documento esteticamente appagante, abbastanza versatile da poter essere adattato ai diversi tipi di target e con le caratteristiche richieste dall'utente attraverso codice immediato e comprensibile.

Dal punto di vista delle prestazioni non è stato ancora eseguito un benchmark,

ma le performance risultano abbastanza promettenti da riuscire a compilare una presentazione di ~30 pagine in 2 secondi su macchine di fascia medio-alta.

Qual è il "documento tipo" di Quarkdown? I requisiti iniziali prevedevano solamente il supporto alle slides, ma si è deciso di puntare ad un approccio molto più generale e versatile.
Il compilatore produce un output universale (uno *snippet* di codice) in un linguaggio target (es. HTML). Il *post-renderer* della pipeline si occupa di adattare questo snippet in un template per il tipo di documento desiderato, come un articolo o una presentazione.
Ciò aggiunge un vantaggio rispetto a tecnologie che si focalizzano su un unico tipo di documento di output.

Qual è il punto di forza? Quarkdown giova di un punto d'incontro tra diverse importanti caratteristiche:

- Si basa su Markdown: è semplice, conciso e familiare a molti;
- Consente una completa possibilità di personalizzazione del documento e della sua paginazione;
- Permette l'automatizzazione tramite il suo supporto allo scripting Turing complete;
- Produce un documento esteticamente gradevole grazie ad un tema tra la selezione offerta.

Qual è il punto debole? Il progetto è giovane e ancora molto acerbo. Diverse funzionalità sono ancora in attesa di essere implementate ed altre sono da perfezionare. È inoltre al momento assente una documentazione o delle guide e la selezione di temi è limitata. Si pianifica di risolvere questi aspetti prossimamente.

15.1 Piani futuri

Ritengo Quarkdown un progetto interessante con la potenzialità di attirare discreta attenzione. Ho intenzione di continuare a sviluppare e mantenere il progetto, aprendolo anche a possibili contribuzioni esterne, e aggiungendo nuove funzionalità, tra cui il rendering in LaTeX, supporto alla creazione di wiki/docs, una maggiore selezione di temi per rendere il risultato più gradevole esteticamente e possibilmente il supporto nativo multipiattaforma, anche sul web.
Un altro importante obiettivo futuro è quello di realizzare una wiki con tutorial e documentazione delle funzioni disponibili.
C'è ancora molto lavoro da fare, ma sono fiducioso che possa diventare un prodotto valido nel lungo periodo.

Bibliografia

[CMS] Commonmark spec. https://spec.commonmark.org/0.31.2.

[GFM] Github flavored markdown. https://github.github.com/gfm.

[HMD] How does a markdown parser work? https://yiou.me/blog/posts/how-does-markdown-parser-work.

[MDG] Markdown guide. https://www.markdownguide.org.

[MKD] Marked. https://github.com/markedjs/marked.

[Nys21] Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021.

Un ringraziamento speciale a chi mi ha accompagnato in questo percorso, e continuerà a farlo.

Alla magnifica esperienza di questi anni, e a tante altre che seguiranno.