PPS24-Declerative Slides

Alex Testa

Indice

1. Introduzione

DeclSlides è un progetto software pensato per semplificare la creazione di presentazioni attraverso un approccio dichiarativo. Il cuore del sistema è un DSL (Domain-Specific Language) sviluppato in Scala, che permette di descrivere una presentazione in modo chiaro, tipizzato ed estendibile, separando la definizione dei contenuti dalla loro effettiva rappresentazione grafica.

1.1 Obiettivo

L'obiettivo principale di DeclSlides è fornire agli utenti un modo semplice e flessibile per creare presentazioni, senza dover affrontare la complessità di strumenti di authoring tradizionali. Grazie alla natura dichiarativa del DSL, gli utenti possono concentrarsi sulla struttura e sui contenuti della presentazione, lasciando a DeclSlides il compito di gestire la formattazione e la generazione del risultato finale.

L'utente può partire da un semplice script con estensione `.sc`, all'interno del quale definire gli elementi principali della presentazione.

Una volta descritta la presentazione, DeclSlides si occupa di trasformarla automaticamente nel formato di output desiderato. In questo modo, lo stesso sorgente può essere riutilizzato per generare rappresentazioni differenti, favorendo la manutenibilità del progetto e riducendo la duplicazione del lavoro.

1.2 Elementi supportati

- **Testo**: Titoli, paragrafi, elenchi puntati, spaziature.
- **Immagini**: Inserimento di immagini.
- **Blocchi di codice**: Visualizzazione di codice sorgente.

1.3 Formati di output supportati

- **HTML**: Generazione di presentazioni web-based, facilmente condivisibili e visualizzabili su qualsiasi dispositivo.
- **Markdown**: Creazione di documenti Markdown, utili per la condivisione su piattaforme come GitHub o per l'integrazione in altri sistemi di documentazione.
- **Testo Semplice**: Esportazione in formato testo, ideale per la stampa o per l'utilizzo in ambienti con limitate capacità di rendering.

1.4 Esempio di utilizzo

Ecco un esempio di come potrebbe apparire un semplice script in DeclSlides:

```
presentation("Hello DeclSlides")
 .use(Theme.default)
 .withFooter("Footer text") {
 deck(
  slide("Intro", Flow) {
   content(
    text("This is the declerative slides tool"),
    text("Here you can write presentations in a declarative way through a DSL in scala"),
    text("The DSL supports: "),
    bullets(
     "Texts",
     "Code snippets",
     "Bullet lists",
     "Spacing",
     "Images"
      )
    )
  },
```

1.5 Contesto di sviluppo

DeclSlides è un progetto accademico sviluppato come parte di un corso universitario.

1.6 Autori

- Alex Testa

2. Dominio

Il dominio applicativo di DeclSlides è quello della definizione e generazione di presentazioni. A differenza degli strumenti di authoring visuale tradizionali, il progetto si colloca nell'intersezione tra presentazioni, automazione e sviluppo software. L'idea di fondo è che una presentazione non sia soltanto un insieme di slide impaginate manualmente, ma possa essere considerata un documento strutturato, descritto formalmente e trasformato in diversi output a partire da una singola sorgente.

In questo contesto, il problema affrontato è duplice. Da un lato, esiste l'esigenza di rappresentare una presentazione in modo ordinato, leggibile e tipizzato. Dall'altro, esiste la necessità di produrre output diversi, ad esempio HTML per la fruizione visuale, testo per debug o ispezione rapida, Markdown per documentazione e condivisione, senza duplicare il contenuto.

Nel dominio di DeclSlides si possono individuare due attori principali, ciascuno con un ruolo distinto all'interno del processo di creazione e fruizione della presentazione.

Il primo attore è **l'autore della presentazione**. Si tratta dello sviluppatore, dello studente o più in generale dell'utente tecnico che realizza la presentazione scrivendo uno script con estensione `.sc` e utilizzando le costrizioni offerte dal DSL. L'autore si occupa quindi di definire la struttura del documento, il titolo, il tema, le slide e i contenuti, concentrandosi soprattutto su cosa deve essere rappresentato, piuttosto che sui dettagli specifici del formato finale.

Il secondo attore è il **consumatore dell'output**, ovvero la persona che usufruisce del risultato prodotto dal sistema. Questo ruolo può corrispondere a chi visualizza la presentazione in formato HTML, a chi legge il file Markdown generato oppure a chi consulta l'output in testo semplice per verificare rapidamente la struttura e i contenuti della presentazione.

In un contesto reale, questi due ruoli possono anche essere ricoperti dalla stessa persona. Ad esempio, l'autore può scrivere lo script, generare la presentazione e poi visualizzarla per controllarne il risultato finale. Tuttavia, dal punto di vista progettuale, DeclSlides mantiene una chiara separazione tra il **formato di authoring**, cioè il modo in cui la presentazione viene descritta tramite DSL, e il **formato di consumo**, cioè il modo in cui la presentazione viene resa disponibile all'utente finale.

Questa separazione rende il sistema più flessibile e manutenibile: lo stesso script sorgente può infatti essere trasformato in formati differenti senza modificare la definizione originale della presentazione.

2.1 Scelte di dominio

In DeclSlides ho scelto di non far costruire direttamente al DSL una presentazione "sempre valida".

Il DSL accumula una descrizione intermedia, mentre la validazione vera rimane nel dominio.

Questa separazione mi ha permesso di mantenere semplice la sintassi utente e, allo stesso tempo, di concentrare le regole in `Presentation`, `Slide` e `SlideElement`.

Il compromesso è che alcuni errori, come una slide vuota o un titolo duplicato, vengono scoperti alla fine della costruzione e non direttamente dal compilatore Scala.

2.2 Entità del dominio

- **Presentation** -> rappresenta l'intera presentazione, con il titolo, il tema, il footer e l'insieme delle slide;
- **Slide** -> rappresenta una singola slide, con un titolo, un layout e un insieme di elementi di contenuto;
- **SlideElement** -> rappresenta un elemento di contenuto all'interno di una slide, che può essere un paragrafo di testo, un elenco puntato, un blocco di codice, uno spazio o un'immagine;
- **Theme** -> rappresenta il tema grafico della presentazione, con colori, font e stili;
- **Layout** -> definisce la disposizione degli elementi all'interno di una slide (Flow, Centered);
- **DomainError** -> rappresenta un errore di validazione del dominio, ad esempio una slide senza titolo o un elemento di contenuto non valido.

2.3 Vincoli di dominio

Le regole principali del dominio riflettono vincoli concreti e comprensibili: il titolo della presentazione non può essere vuoto, una presentazione deve avere almeno una slide, i titoli delle slide devono essere univoci, una slide deve contenere almeno un elemento, testi e blocchi di codice non possono essere vuoti, le immagini devono avere sorgente e testo alternativo significativi, il footer, se presente, non può essere privo di contenuto.

Questa attenzione ai vincoli è centrale, perché sposta il controllo di correttezza dal momento finale del rendering alla costruzione stessa del modello. In altre parole, il sistema non tratta l'errore come un evento eccezionale da nascondere, ma come una parte normale del processo di authoring che va rappresentata in modo esplicito.

2.4 Tabella concettuale del dominio

Entità	Ruolo	Responsabilità principale
Presentation	radice del modello	definisce titolo, tema, footer e insieme di slide
Slide	unità di contenuto	definisce titolo, layout e contenuti
SlideElement	componente di slide	rappresenta testo, liste, codice, immagini e spaziatori
Theme	configurazione visuale	definisce palette cromatica e identità visiva
Layout	disposizione	suggerisce la disposizione dei contenuti
DomainError	validazione	descrive errori strutturali e semantici del modello

2.5 Diagramma del dominio

```
classDiagram
    class Presentation {
      +title: String
      +theme: Theme
      +footer: Option[String]
      +slides: Vector[Slide]
    }

    class Slide {
      +title: String
      +layout: Layout
      +elements: Vector[SlideElement]
    }

    class SlideElement {
      <<enum>>
      Paragraph
      BulletList
      CodeBlock
      Spacer
      Image
    }

    class Theme {
      +name: String
      +background: String
      +foreground: String
      +accent: String
      +codeBackground: String
    }

    class Layout {
      <<enum>>
      Flow
      Centered
    }

    Presentation "1" --> "*" Slide
    Slide "1" --> "*" SlideElement
    Presentation "1" --> "1" Theme
    Slide "1" --> "1" Layout
```

rappresentazione concettuale delle principali entità del dominio e delle loro relazioni. Il diagramma mette in evidenza la centralità del modello `Presentation`*, la composizione delle slide e la natura tipizzata dei contenuti.*

3. Obiettivi del progetto

Gli obiettivi del progetto sono stati definiti in modo da tenere insieme prodotto, qualità del software e valore formativo. Non si è trattato soltanto di costruire “qualcosa che funzioni”, ma di realizzare un sistema piccolo, coerente e progettato con metodo.

3.1 Caratteristiche di base

- Definizione di una presentazione composta da una sequenza di slide;
- Creazione di slide tramite costrutti dichiarativi della DSL
- Supporto per diversi tipi di contenuto all’interno delle slide (testo, liste puntate, blocchi di codice, immagini);
- Composizione e riutilizzo di componenti di slide tramite funzioni e astrazioni del linguaggio;
- Rappresentazione interna della presentazione tramite una struttura dati immutabile;
- Generazione di una rappresentazione visualizzabile della presentazione tramite un sistema di rendering (ad esempio HTML o formato testuale).

3.2 Caratteristiche aggiuntive

- Rendering delle slide tramite interfaccia a linea di comando (CLI): l’utente potrà fornire il file contenente la definizione della presentazione e ottenere automaticamente il rendering delle slide.
- Supporto a diversi formati di output (ad esempio HTML, Markdown).
- Sviluppo di una interfaccia grafica per l’editing delle slide.

3.3 Obiettivi funzionali

Il primo gruppo di obiettivi riguarda **ciò che il sistema deve fare concretamente**.

Il progetto doveva permettere di **definire una presentazione in modo dichiarativo**, tramite un DSL leggibile e naturale. Questa scelta nasce dalla volontà di ridurre il rumore sintattico e di fare in modo che il codice della presentazione assomigli il più possibile alla sua struttura concettuale. L’autore deve poter esprimere titolo, tema, footer, slide, layout e contenuti con una sintassi chiara e coerente.

Un secondo obiettivo funzionale era la **renderizzazione multi-formato**. La stessa presentazione deve poter essere trasformata in HTML, testo semplice e Markdown, senza duplicazione dei contenuti. Questo obiettivo è importante perché dimostra che il modello di dominio non è accoppiato a una singola vista o tecnologia di output.

Un terzo obiettivo era fornire una **CLI utilizzabile a partire da un file** `.sc`, in modo che il progetto potesse essere usato anche fuori dal codice sorgente interno, come strumento vero e proprio.

3.4 Obiettivi tecnici

Sul piano tecnico, l’obiettivo principale era **costruire una soluzione con una separazione netta delle responsabilità**. Il dominio doveva restare puro e indipendente dai dettagli di rendering; la DSL doveva restare espressiva ma non diventare il luogo in cui si annidano i vincoli; la CLI doveva orchestrare, non contenere logica di dominio.

Un altro obiettivo tecnico era **garantire estendibilità**. L’aggiunta progressiva di immagini, footer, renderer Markdown e sito di presentazione mostra che il progetto è stato pensato per evolvere senza richiedere continue riscritture strutturali.

Infine, un obiettivo essenziale era la **testabilità**. Molte parti del sistema sono state progettate in modo da poter essere isolate e verificate direttamente, dalla validazione del dominio al parsing CLI, fino ai singoli renderer.

3.5 Obiettivi di qualità

DeclSlides doveva essere **leggibile, coerente e manutenibile**. Questo significa evitare magic numbers, ridurre l'opacità del codice, esplicitare i concetti con nomi significativi, centralizzare i messaggi e preferire modelli tipizzati ai valori grezzi.

La qualità non è stata intesa come un attributo astratto, ma come una somma di pratiche concrete: refactoring, TDD, documentazione, test, CI e commit convenzionali.

Le performance attualmente non ricoprono un obiettivo primario.

3.6 Obiettivi formativi

Dal punto di vista formativo, il progetto doveva rappresentare un caso realistico in cui applicare principi di ingegneria del software. In particolare: - modellazione di dominio; - separazione di responsabilità; - costruzione di un DSL; - progettazione a layer; - validazione esplicita tramite `Either` ed errori tipizzati; - sviluppo guidato dai test(TDD); - integrazione con pipeline CI/CD; - documentazione tecnica strutturata.

3.7 Obiettivi opzionali

Alcuni obiettivi possono essere considerati opzionali, nel senso che rappresentano funzionalità aggiuntive che arricchiscono il progetto ma non sono essenziali per la sua validità. Tra questi:

- Supporto a temi personalizzati, con definizione di palette cromatiche e stili;
- Implementazione di un sistema di layout più sofisticato, con posizionamento libero degli elementi
- Sviluppo di una GUI per l'editing visuale delle presentazioni, con anteprima in tempo reale;
- Performance ottimizzata per presentazioni di grandi dimensioni, con caching e rendering incrementale;

4. Processo di sviluppo

Il processo di sviluppo adottato è stato di tipo iterativo, con una forte attenzione alla progressione incrementale e alla qualità continua. Il progetto non è stato costruito in un solo passaggio, ma attraverso piccoli step, ognuno dei quali ha portato una funzionalità, un refactor o un miglioramento alla struttura esistente.

L'approccio seguito è stato **Scrum-inspired**, ma alleggerito e adattato a un progetto tecnico di dimensione contenuta. In pratica, il lavoro è stato organizzato in task ben definiti, con cicli di implementazione basati su obiettivi chiari: prima introdurre una funzionalità tramite test, poi implementare il codice minimo necessario, infine rifinire con refactoring e pulizia architetturale.

Una parte significativa del lavoro è stata svolta con **TDD** o comunque con una forte **priorità ai test**. Questo ha permesso di rendere esplicite le aspettative su DSL, dominio, renderer e CLI prima di consolidare l'implementazione. L'effetto più importante non è stato solo l'aumento della copertura, ma la definizione più chiara del comportamento desiderato.

Dal punto di vista collaborativo, `Git` è stato usato come strumento centrale di coordinamento, con un'attenzione specifica ai commit piccoli, descrittivi e allineati al principio di tracciabilità. L'uso di **Conventional Commits** ha contribuito a mantenere leggibile la storia del progetto e a supportare il rilascio automatizzato tramite `semantic-release`.

Infine si è scelto di utilizzare la piattaforma `CodeCov` per monitorare la copertura dei test, con l'obiettivo di mantenere un alto standard qualitativo e di identificare rapidamente eventuali aree non sufficientemente testate. Questo ha contribuito a garantire che ogni nuova funzionalità fosse accompagnata da un adeguato set di test, rafforzando la stabilità complessiva del progetto.

4.1 Tabella del processo

Aspetto	Scelta Adottata	Motivazione
Gestione del lavoro	Approccio Iterativo	permette di sviluppare per incrementi, con feedback continuo
Sviluppo funzionalità	TDD/test-first	chiarisce il comportamento atteso e riduce regressioni
Versionamento	Git	supporta collaborazione, rollback e tracciabilità
Conventional Commit	Conventional Commits	migliora leggibilità della history e release automation
Revisione	Scalafmt & Scalafix	garantiscono coerenza stilistica e qualità del codice
Qualità continua	CI (GithubActions)	automatizza test, linting e code coverage
Rilascio	Semantic-Release	automatizza il processo di rilascio basato sui commit e sulla semantica delle versioni e di pubblicazione dell'artefatto
Docs	Mkdocs (GithubPages)	facilita la creazione e la pubblicazione di documentazione accessibile e ben strutturata

4.2 Requisiti

La definizione dei requisiti è stata fondamentale per dare direzione al progetto e per collegare le scelte di design al valore reale prodotto. I requisiti sono stati organizzati distinguendo tra necessità di business, requisiti funzionali, requisiti non funzionali, estensioni opzionali e possibili sviluppi futuri.

4.2.1 Requisiti di Business

Dal punto di vista del valore d'uso, il sistema doveva permettere di trattare una presentazione come artefatto software, e non solo come documento visuale. Questo implica che il contenuto debba essere:

- Versionabile;
- Riproducibile;
- Leggibile nel tempo;
- Validabile;
- Renderizzabile in più formati.

Un secondo requisito di business consisteva nel poter usare il progetto sia come strumento pratico sia come oggetto di studio ingegneristico, con un'attenzione forte alla struttura del codice e alla dimostrazione delle scelte architetturali.

4.2.2 Requisiti Funzionali

Codice	Descrizione	Area	Stato	Note
RF-01	Definire una presentazione tramite DSL	DSL	Implementato	Sintassi dichiarativa tramite Scala 3
RF-02	Definire slide con titolo e layout	DSL/Domain	Implementato	Supporto a layout `Flow` e `Centered`
RF-03	Inserire paragrafi di testo	DSL/Domain	Implementato	Validazione su contenuto non vuoto
RF-04	Inserire liste puntate	DSL/Domain	Implementato	Validazione della lista e sui singoli item
RF-05	Inserire blocchi di codice	DSL/Domain	Implementato	Con linguaggio sorgente
RF-06	Inserire spaziature	DSL/Domain	Implementato	
RF-07	Inserire immagini	DSL/Domain	Implementato	Con URL e Alt text
RF-08	Configurare un tema	DSL/Domain	Implementato	`Default`, `Dark`, `Conference`
RF-09	Configurazione di un footer	DSL/Domain	Implementato	Footer opzionale globale
RF-10	Renderizzare in HTML	Renderer	Implementato	Layout full screen e navigazione
RF-11	Renderizzare in Testo semplice	Renderer	Implementato	Utile ai fini di debug e ispazione
RF-12	Renderizzare in Markdown	Renderer	Implementato	Utile per documentazione
RF-13	Fornire una CLI per renderizzare	CLI/Application	Implementato	Supporto a input `.sc` da file e output su file
RF-14	Validare in modo esplicito gli errori di dominio	Domain	Implementato	Errori tipizzati con messaggi chiari
RF-15	Fornire tutta la documentazione	Docs	Implementato	Documentare tutto il progetto a fini conoscitivi

4.2.3 Requisiti Non Funzionali

Usabilità

Il DSL doveva essere leggibile e vicino alla struttura logica di una presentazione. La CLI doveva essere semplice da usare e basata su parametri espliciti (`--input` , `--format` , `--output`).

Manutenibilità

La codebase doveva essere organizzata in moduli e package coerenti, con responsabilità ben separate. L'aggiunta di nuove funzionalità non doveva richiedere modifiche pervasive e incontrollate.

Testabilità

Il sistema doveva consentire test unitari e di integrazione mirati, con componenti facilmente isolabili. Le funzioni di parsing, i renderer e la validazione del dominio dovevano poter essere verificati senza dipendere dall'intero sistema.

Estendibilità

Il progetto doveva essere predisposto all'aggiunta di nuovi renderer, nuovi elementi di slide e nuove opzioni del DSL.

Portabilità

La pipeline CI doveva verificare il comportamento su più sistemi operativi, così da evitare accoppiamenti impliciti con un solo ambiente.

Documentazione

Era richiesto che il progetto fosse accompagnato da una documentazione leggibile, sia per l'uso, sia per la comprensione architetturale.

4.2.4 Requisiti Opzionali

- Sito di presentazione pubblicato automaticamente;
- GUI per la creazione e modifica di presentazioni.

4.2.5 Sviluppi Futuri

- Supporto a temi personalizzati definiti dall'utente;
- Supporto a layout aggiuntivi (es. griglia, split);
- Supporto a elementi multimediali (video, audio);
- Supporto a formati di output aggiuntivi (es. PDF, PowerPoint);

5. Stack Tecnologico

Le tecnologie adottate non sono state scelte in modo decorativo, ma in relazione diretta alle esigenze del progetto.

Tecnologia	Ruolo nel progetto	Motivazione
Scala 3	Linguaggio di programmazione principale	offre ottimo supporto a DSL, immutabilità, ADT, enum, espressività tipizzata
Sbt	Build Tool	standard nell'ecosistema Scala, adatto a test e build
Scala CLI	Esecuzione script utente	permette di valutare `.sc` in modo semplice e integrabile
ScalaTags	Generazione HTML	consente di costruire HTML tipizzato e leggibile senza template engine esterno
GitHub Actions	CI/CD	automatizza test e build, integrato con GitHub
os-lib	File system e processi	semplifica I/O, path handling e chiamate di processo in modo chiaro
ScalaTest	Testing	API leggibile e coerente con lo stile del progetto
Semantic-Release	Release automation	automatizza versionamento e pubblicazione dell'artefatto
GitHub Pages	Pubblicazione del sito	soluzione leggera per ospitare il sito del progetto
Tailwind CSS via CDN	Stile visuale HTML	permette di ottenere rapidamente una resa moderna senza build frontend dedicata
Commit Lint	Linting dei commit	garantisce coerenza nei messaggi di commit, facilitando la gestione del progetto e l'automazione delle release
Scala Fmt	Formattazione del codice	assicura uno stile di codice uniforme e leggibile
Scala Fix	Refactoring automatico	aiuta a mantenere il codice pulito e aggiornato con le best practice di Scala
Husky	Git Hooks	automatizza l'esecuzione di script prima dei commit e dei push, garantendo la qualità del codice e la coerenza dei commit
Codecov	Monitoraggio della copertura dei test	fornisce metriche dettagliate sulla copertura del codice, aiutando a identificare aree non sufficientemente testate
Mkdocs	Documentazione	facilita la creazione e la pubblicazione di documentazione accessibile e ben strutturata
Mermaid	Diagrammi	consente di creare diagrammi di classe e altri tipi di visualizzazioni direttamente nei documenti Markdown
Renovate	Gestione delle dipendenze	automatizza l'aggiornamento delle dipendenze, mantenendo il progetto sicuro e aggiornato

5.1 Motivazione delle scelte

5.1.1 Scala 3

Scala 3 è stata una scelta naturale per un progetto che ruota attorno a un DSL. La possibilità di modellare il dominio con `enum`, `case class`, `pattern matching` e `funzioni di alto livello` ha reso più agevole costruire una sintassi espressiva senza compromettere il rigore.

5.1.2 Sbt

Sbt è lo standard de facto per i progetti Scala. La sua flessibilità e il supporto per la gestione delle dipendenze, i test e la compilazione incrementale lo rendono ideale per un progetto che richiede iterazione rapida e affidabilità.

5.1.3 Scala CLI

La presenza di una CLI che esegue script utente rende Scala CLI uno strumento molto coerente con il problema. Invece di inventare un interprete ad hoc, il sistema sfrutta un'infrastruttura già esistente per valutare la descrizione della presentazione.

5.1.4 ScalaTags

Per il renderer HTML si è preferito evitare un motore di template tradizionale. ScalaTags si integra bene con Scala, permette una costruzione programmatica del DOM e mantiene il rendering vicino alla logica che lo governa aumentando l'espressività.

5.1.5 Tailwind CSS via CDN

Per il sito e per l'HTML è stata privilegiata la semplicità operativa. L'uso del CDN elimina una pipeline frontend dedicata e accelera l'iterazione grafica. È una scelta consapevole, adatta a questa scala progettuale.

5.1.6 Commit Lint

L'adozione di Commit Lint è stata motivata dalla necessità di mantenere una storia dei commit chiara e coerente. Questo non solo facilita la collaborazione, ma è anche essenziale per l'automazione delle release con Semantic-Release, che si basa sui messaggi di commit per determinare le versioni da rilasciare.

5.1.7 Scala Fmt e Scala Fix

L'uso di Scala Fmt e Scala Fix è stato dettato dalla volontà di mantenere un codice pulito, leggibile e conforme alle best practice. Scala Fmt assicura uno stile uniforme, mentre Scala Fix aiuta a refactorare il codice in modo automatico, mantenendo il progetto aggiornato con le evoluzioni del linguaggio e delle librerie utilizzate.

5.1.8 Husky

Husky è stato scelto per automatizzare l'esecuzione di script prima dei commit e dei push. Questo garantisce che i test vengano eseguiti, che i commit siano formattati correttamente e che i messaggi di commit rispettino le convenzioni stabilite, contribuendo a mantenere la qualità del codice e la coerenza del progetto.

5.1.9 Codecov

L'adozione di Codecov è stata motivata dalla necessità di monitorare la copertura dei test in modo dettagliato. Codecov fornisce metriche precise sulla copertura del codice, aiutando a identificare aree che potrebbero essere insufficientemente testate e a mantenere un alto standard qualitativo per il progetto.

5.1.10 Mkdocs

Mkdocs è stato scelto per la documentazione del progetto grazie alla sua semplicità d'uso e alla capacità di generare siti statici ben strutturati. La possibilità di scrivere la documentazione in Markdown e di ospitarla facilmente su GitHub Pages lo rende una soluzione ideale per rendere accessibile e ben organizzata la documentazione di DeclSlides.

5.1.11 Mermaid

Mermaid è stato adottato per la creazione di diagrammi direttamente nei documenti Markdown. Questo strumento consente di rappresentare visivamente concetti complessi come la struttura del dominio o l'architettura del sistema in modo chiaro e integrato con la documentazione, migliorando la comprensione e la comunicazione delle idee chiave del progetto.

5.1.12 Renovate

Renovate è stato scelto per automatizzare la gestione delle dipendenze del progetto. Questo strumento monitora le dipendenze utilizzate e crea pull request automatiche quando sono disponibili aggiornamenti, contribuendo a mantenere il progetto sicuro e aggiornato senza richiedere un intervento

manuale costante. Renovate supporta anche la configurazione di regole personalizzate per gestire le dipendenze in modo flessibile, adattandosi alle esigenze specifiche del progetto.

6. Architettura Generale

L'architettura di DeclSlides è organizzata in layer distinti, con una separazione chiara fra modello, definizione del contenuto, orchestrazione e rappresentazione finale. Lo stile architetturale adottato può essere descritto come una forma leggera di layered architecture, con forti elementi di separazione del dominio e responsabilità ben delimitate (simile al Domain Driven Design).

Il cuore del sistema è il dominio, che modella il concetto di presentazione e quali regole debba rispettare. Intorno a questo nucleo si collocano il DSL, che fornisce una sintassi espressiva per costruire il modello.

L'application layer orchestra, validazione, bootstrap ed esecuzione.

I renderer, che traducono il modello in output.

La CLI, che espone il sistema verso l'esterno.

```
flowchart LR
    User[Utente / Autore] --> Script[Script .sc con DSL]
    Script --> CLI[CLI declslides]
    CLI --> App[Application Layer]
    App --> DSL[DSL]
    DSL --> Domain[Domain Model]
    App --> Runner[Scala CLI Script Runner]
    Runner --> Domain
    Domain --> Rendering[Rendering Layer]
    Rendering --> HTML[HTML Renderer]
    Rendering --> TEXT[Text Renderer]
    Rendering --> MD[Markdown Renderer]
    HTML --> Output[File di output]
    TEXT --> Output
    MD --> Output
```

Visione d'insieme dell'architettura di DeclSlides. Il diagramma evidenzia la distinzione tra definizione della presentazione, validazione del modello e traduzione in formati diversi.

Il diagramma mostra il flusso principale: l'utente scrive uno script .sc, la CLI lo inoltra al livello applicativo, il runner lo esegue, il dominio viene validato e infine il modello risultante viene passato al renderer corrispondente per produrre l'output.

6.1 Stile architetturale adottato

La scelta di una layered architecture è stata motivata dall'esigenza di mantenere distinguibili:

- Regole del dominio;
- Sintassi del DSL;
- Logica di orchestrazione;
- Dettagli del rendering;
- Interfaccia utente (CLI).

Questo approccio ha ridotto l'accoppiamento e ha facilitato l'estensione del sistema. La prova più evidente è che nel corso dello sviluppo si sono potuti aggiungere footer, immagini e renderer Markdown senza dover riscrivere il nucleo dell'applicazione.

6.2 Organizzazione del codice

La struttura del progetto è stata organizzata per comunicare chiaramente le responsabilità dei diversi moduli logici.

Modulo	Responsabilità
`declslides.domain`	Modello di dominio e validazione
`declslides.dsl`	Definizione del DSL
`declslides.application`	Orchestrazione, validazione e bootstrap
`declslides.cli`	Interfaccia a riga di comando
`declslides.rendering`	Contratto dei renderer e registry
`declslides.rendering.html`	Implementazione del renderer HTML
`declslides.rendering.text`	Implementazione del renderer Testuale
`declslides.rendering.markdown`	Implementazione del renderer Markdown
`declslides.utils`	Funzioni di utilità condivise
`src/test`	Suite di test
`/site`	Sito di presentazione del progetto
`.github/workflows`	Workflow di CI/CD
`/readme`	Script per documentazione del README
`/examples`	Esempi di script .sc

Il diagramma seguente mostra le dipendenze fra i moduli principali, evidenziando la separazione fra dominio, definizione del contenuto, orchestrazione e rappresentazione finale.

```
flowchart TB
    CLI["declslides.cli"] --> APP["declslides.application"]
    CLI --> RENDERING["declslides.rendering"]

    APP --> RENDERING
    APP --> UTILS["declslides.utils"]

    DSL["declslides.dsl"] --> DOMAIN["declslides.domain"]

    RENDERING --> DOMAIN
    RENDERING --> HTML["declslides.rendering.html"]
    RENDERING --> TEXT["declslides.rendering.text"]
    RENDERING --> MD["declslides.rendering.markdown"]

    HTML --> DOMAIN
    HTML --> RENDERING
    HTML --> UTILS

    TEXT --> DOMAIN
    TEXT --> RENDERING

    MD --> DOMAIN
    MD --> RENDERING
```

7. Design di dettaglio

7.1 Modello di dominio

Il modello di dominio è composto principalmente da `Presentation`, `Slide`, `SlideElement`, `Theme`, `Layout` e `DomainError`. La scelta di tenere Presentation e Slide come modelli validati è stata molto importante: ciò che esce dal dominio non è semplicemente "dato strutturato", ma un oggetto semanticamente consistente.

Il fatto che `Presentation.apply(...)` e `Slide.apply(...)` restituiscano `Either[Vector[DomainError], ...]` è una scelta significativa: gli errori non vengono lanciati come eccezioni, ma espressi come parte del flusso applicativo. Questo rende il sistema più prevedibile e più adatto a uno stile funzionale.

7.2 Logica applicativa

Il livello applicativo coordina la risoluzione del formato, il caricamento dello script, la generazione del bootstrap e l'esecuzione del processo tramite Scala CLI. Qui si trovano concetti come `RenderCommand`, `ScalaCliScriptRunner`, `InputScriptValidator`, `InputSourceReader` e `BootstrapSourceFactory`.

L'application layer non implementa le regole del dominio e non produce direttamente il rendering: il suo compito è comporre i servizi e orchestrare il flusso.

7.3 Interfaccia Utente

L'interfaccia principale del progetto è una CLI. La sua responsabilità è molto chiara: ricevere argomenti, validare l'invocazione, costruire il comando appropriato, gestire errori user-facing e fornire un feedback finale. In parallelo, il progetto include anche una UI web statica per la presentazione del DSL stesso.

7.4 Persistenza o gestione dello stato

Non essendoci una persistenza classica, il progetto gestisce lo stato in memoria. Tuttavia, esiste una nozione di stato importante nel DSL: `PresentationState` e `SlideState` rappresentano lo stato intermedio della costruzione dichiarativa. Questo stato è immutabile e trasformato funzionalmente.

7.5 Servizi, controller o manager

Nel progetto non compaiono controller in senso MVC tradizionale, ma sono presenti componenti di coordinamento:

- `RenderCommand` funge da coordinatore per il processo di rendering.
- `ScalaCliScriptRunner` è un servizio che incapsula la logica di esecuzione dello script.
- `RendererRegistry` è un servizio che gestisce la registrazione e risoluzione dei renderer disponibili.
- `CliProgram` è un controller che gestisce l'interazione con la CLI e coordina l'invocazione del processo di rendering.

Questi componenti gestiscono flussi, non regole di business.

7.6 Utilità e helper

Sono stati introdotti piccoli componenti riutilizzabili per:

- Messaggi di errore standardizzati (`ErrorMessages`);
- Caricamento di resource testuali (`ResourceLoader`);
- Generazione del boostrap per Scala CLI (`BootstrapSourceFactory`).
- Registry dei render disponibili (`RendererRegistry`).
- Messaggi di successo standardizzati (`SuccessMessages`).

La loro presenza riduce la duplicazione e facilità di refactoring.

7.7 Funzionalità principali

7.7.1 Funzionalità 1: Definizione dichiarativa di una presentazione

Descrizione

L'utente può descrivere una presentazione tramite un DSL in Scala, costruendo il deck attraverso chiamate leggibili e composizionali. Questo è il cuore del progetto, perché separa il contenuto dalla rappresentazione finale.

Flusso

1. L'utente scrive uno script .sc utilizzando il DSL.
2. Definisce titolo, tema, footer e slide.
3. Ogni slide è costruita con elementi come testo, immagini, liste, ecc.
4. La descrizione viene passata al dominio, che la valida e costruisce un modello di presentazione disponibile al renderer.

Componenti coinvolti

- `DSL`
- `Presentation`
- `Slide`
- `SlideElement`
- `Theme`
- `Layout`
- `DomainError`

Requisiti soddisfatti

- RF-01
- RF-02
- RF-03
- RF-04
- RF-05
- RF-06
- RF-07
- RF-08
- RF-09

Esempio

```
presentation("DeclSlides Demo")
  .use(Theme.conference)
  .withFooter("Alex Testa") {
    deck(
      slide("Intro") {
        content(
          text("DeclSlides turns code into presentations."),
          bullets(
            "Typed DSL",
            "Multiple renderers",
            "CLI execution"
          )
        )
      },
      slide("Code") {
        content(
          code(
            "scala",
            """println("hello declslides")"""
          )
        )
      }
    )
  }
```

7.7.2 Funzionalità 2: Rendering multi-formato

Descrizione

La stessa presentazione può essere esportata in HTML, testo semplice e Markdown. Questo dimostra che il sistema modella il contenuto in modo indipendente dal formato di destinazione.

Flusso

1. Il formato richiesto viene risolto dal registry.
2. Il renderer appropriato viene selezionato.
3. La presentazione validata viene convertita in `Document`.
4. Il `Document` viene serializzato nel formato desiderato e scritto su file.

Componenti coinvolti

- `RendererRegistry`
- `HtmlRenderer`
- `TextRenderer`
- `MarkdownRenderer`
- `RenderCommand`

7.8 Requisiti soddisfatti

- RF-10
- RF-11
- RF-12

7.8.1 Funzionalità 3: Esecuzione CLI

Descrizione

Il progetto include una CLI che consente di passare un file `.sc`, scegliere il formato di output e generare il documento finale.

Flusso

1. L'utente invoca il jar o il comando CLI.
2. La CLI valida gli argomenti e costruisce un `RenderCommand`.
3. Il `RenderCommand` viene eseguito, orchestrando il processo di rendering.
4. Il runner genera il boostrap ed esegue lo script tramite Scala CLI.
5. Il risultato viene scritto su file e un messaggio di successo viene mostrato.

Componenti coinvolti

- `CliProgram`
- `DeclslidesCLI`
- `CliArugumentParser`
- `RenderCommandFactory`
- `ScalaCliScriptRunner`
- `BootstrapSourceFactory`

Requisiti soddisfatti

- RF-13

Esempio

```
java -jar declslides.jar --input slides.sc --format html --output deck.html
```

7.8.2 Funzionalità 4: Validazione e gestione degli errori

Descrizione

Il sistema valida la presentazione durante la costruzione del modello di dominio e gestisce gli errori in modo funzionale, restituendo messaggi chiari all'utente.

Flusso

1. Durante la costruzione di `Presentation` e `Slide`, eventuali errori vengono raccolti in un `Vector[DomainError]`.
2. Se la validazione fallisce, il processo si interrompe e i messaggi di errore vengono mostrati all'utente.
3. Se la validazione ha successo, il processo continua normalmente.

Componenti coinvolti

- `Presentation`
- `Slide`
- `DomainError`
- `RenderCommand`
- `CliProgram`

Requisiti soddisfatti

- RF-14

7.8.3 Funzionalità 5: Documentazione e sito web

Descrizione

Il progetto include un sito statico in inglese che presenta il DSL, mostra esempi di utilizzo, spiega il flusso di compilazione e viene pubblicato automaticamente tramite GitHub Actions.

Sito di presentazione del DSL

Componenti coinvolti

- `/site`
- Workflow di GitHub Actions per la pubblicazione
- GitHub Pages

Requisiti soddisfatti

- RF-15

7.9 Flussi principali

```
flowchart TD
    A[Utente invoca la CLI] --> B[Parsing argomenti]
    B --> C[Creazione configurazione valida]
    C --> D[Risoluzione formato]
    D --> E[Validazione input script]
    E --> F[Lettura script]
    F --> G[Generazione bootstrap]
    G --> H[Esecuzione tramite Scala CLI]
    H --> I[Costruzione Presentation]
    I --> J[Validazione dominio]
    J --> K[Renderer selezionato]
    K --> L[Scrittura file output]
```

Questo è il flusso operativo più importante del progetto. Ogni passaggio ha una responsabilità chiara, e i punti di possibile fallimento vengono gestiti in modo esplicito con errori leggibili.

7.9.1 Sequence diagram del caso d'uso principale

```
sequenceDiagram
    participant U as Utente
    participant CLI as DeclSlides CLI
    participant APP as Application Layer
    participant RUN as ScalaCliScriptRunner
    participant REG as RendererRegistry
    participant REN as Renderer
    participant FS as File System

    U->>CLI: --input slides.sc --format html --output deck.html
    CLI->>APP: run(request)
    APP->>REG: resolve("html")
    REG-->>APP: HtmlRenderer target
    APP->>RUN: render(input, target, output)
    RUN->>FS: read script
    RUN->>RUN: generate bootstrap
    RUN->>RUN: execute Scala CLI
    RUN->>REN: render(presentation)
    REN->>FS: write output file
    RUN-->>APP: success
    APP-->>CLI: success
    CLI-->>U: confirmation message
```

7.9.2 Flusso di errore

Un aspetto importante è il flusso di errore. Se la presentazione contiene contenuti invalidi, il sistema non produce un output parziale e opaco, ma restituisce un insieme di errori espliciti. Questo comportamento rende il progetto più affidabile e più utile in un contesto tecnico.

7.10 Scelte progettuali

7.10.1 Separazione tra dominio e DSL

Il DSL è stato progettato come un livello di authoring, non come il luogo in cui vivono le regole del sistema. La validazione resta nel dominio, perché è il dominio a dover definire cosa sia una presentazione corretta. Questa scelta evita che i vincoli siano sparsi in funzioni di costruzione e rende il modello più robusto.

Il punto di forza risiede nella capacità di costruire un modello di dominio ricco e validato, che poi può essere trasformato in qualsiasi formato desiderato. Il DSL è solo una comoda interfaccia per costruire quel modello, ma non è il cuore del sistema.

7.10.2 Error handling esplicito

Invece di lanciare eccezioni per i casi normali di invalidità, il progetto usa `Either` e tipi di errore dedicati. Questo vale sia nel dominio sia in varie parti dell'application layer.

Questa scelta migliora testabilità e leggibilità, perché rende espliciti i punti di fallimento e riduce il rischio di comportamento implicito.

7.10.3 Introdurre un registry dei renderer

Il `RendererRegistry` separa la conoscenza dei renderer disponibili dal loro utilizzo. Questa scelta è piccola ma importante, perché consente di aggiungere nuovi renderer senza cambiare il contratto generale del sistema.

7.10.4 HTML generato in modo funzionale

Per il renderer HTML si è scelto di usare ScalaTags, evitando un sistema di template esterno. Questo rende il rendering più vicino alla struttura del dominio e mantiene tutto nel linguaggio principale del progetto.

7.10.5 CLI come layer sottile

La CLI non interpreta direttamente il file `.sc`: genera un piccolo bootstrap e lo esegue tramite Scala CLI.

Questa soluzione ha ridotto la complessità interna del progetto, perché evita di implementare un parser o un compilatore custom.

Il costo è una dipendenza operativa in più: l'utente deve avere Scala CLI disponibile e la variabile `DECLSLIDES_SCALA_CLI` configurata correttamente.

Per questo motivo considero la CLI funzionante, ma non ancora ideale dal punto di vista della distribuzione.

7.11 Pattern e principi di buona programmazione

7.11.1 Separazione delle responsabilità (Single Responsibility Principle)

Il progetto è stato costruito applicando con costanza il principio di separazione delle responsabilità. Ogni package ha un ruolo chiaro e ogni componente tende a fare una sola cosa.

Molti dei refactor introdotti nel corso dello sviluppo vanno proprio in questa direzione. Esempi evidenti sono:

- `CliArgumentParser` separato dalla CLI.
- `BootstrapSourceFactory` separato dal runner.
- `InputSourceLoader` e `InputScriptValidator` separati.
- `RenderFormatResolver` separato dal comando di rendering.

7.11.2 Factory

Le factory compaiono in modo esplicito soprattutto nella creazione dei comandi e nella generazione del bootstrap.

7.11.3 Strategy

I renderer costituiscono una forma chiara di Strategy: a partire da un `RenderFormat`, il sistema seleziona la strategia di rendering appropriata.

7.11.4 Builder

Il DSL costruisce progressivamente il modello della presentazione tramite `PresBuild` e `SlideBuild`. Questa struttura ricorda un builder funzionale, basato su trasformazioni di stato immutabile.

7.11.5 Immutabilità

La presenza di state object immutabili e di modelli dichiarativi immutabili ha ridotto il rischio di bug e ha reso il flusso più prevedibile.

7.11.6 Open/Closed Principle

L'aggiunta di nuovi elementi come immagini e footer, o di un nuovo renderer Markdown, mostra che il sistema è stato esteso senza modificare in modo distruttivo il nucleo del progetto.

7.12 Gestione dello stato

Il progetto non utilizza un database né una persistenza applicativa nel senso classico, ma gestisce uno stato significativo durante la costruzione della presentazione e nella produzione dell'output.

Il DSL usa due forme di stato intermedio:

- `PresentationState`
- `SlideState`

Entrambi rappresentano lo stato "in costruzione" e vengono trasformati in modo puro, tramite funzioni composite. Questa soluzione è particolarmente adatta a un DSL, perché permette di accumulare configurazioni e contenuti mantenendo un modello semplice.

I dati centrali sono:

- titolo della presentazione;
- tema;
- footer;
- elenco delle slide;
- layout per ogni slide;
- sequenza ordinata di elementi.

La validazione interviene quando lo stato intermedio viene trasformato nel modello finale. Questo è un punto chiave: il DSL non impedisce di descrivere tutto, ma è il dominio a decidere se il risultato è valido.

7.13 Testing e qualità

La strategia di testing del progetto è stata una delle leve principali di qualità. Il test non è stato trattato come verifica finale, ma come strumento di progettazione e chiarificazione del comportamento atteso.

L'adozione del TDD in molte feature ha portato benefici concreti. Prima si è descritto il comportamento desiderato attraverso test leggibili, poi si è introdotto il codice minimo per farli passare, infine si è rifinita la struttura con refactor. Questo ha reso più sicuro introdurre nuove funzionalità come immagini, footer e renderer Markdown.

7.13.1 Tabella di copertura qualitativa

Tipo	Area di copertura	Strumento
Unit Test	Dominio	ScalaTest
Unit Test	DSL	ScalaTest
Unit Test	Render	ScalaTest
Unit Test	CLI	ScalaTest
Integration Test	Application	ScalaTest
Test CI multi-OS	Intero progetto	GithubActions

Qualità del codice

La qualità del codice viene supportata da:

- Linter
- Formatter
- Conventional Commits
- CI Automatica
- Documentazione tramite Scaladoc

7.14 CI/CD, build e deploy

Il progetto dispone di una pipeline CI/CD basata su GitHub Actions. La pipeline è organizzata in fasi distinte e coerenti:

1. Test su più sistemi operativi e pubblicazione della coverage solo su macchina Linux;
2. Build su più sistemi operativi;
3. Release automatizzata sul branch `main`;
4. Pubblicazione del sito su GithubPages con documentazione annessa.

```
flowchart LR
    A[Push / Pull Request] --> B[Test multi-OS & Publish Coverage]
    B --> C[Build multi-OS]
    C --> D[Release su main]
    D --> E[semantic-release]
    E --> F[Deploy sito su GitHub Pages]
```

rappresentazione della pipeline di integrazione e rilascio continuo del progetto. La struttura mostra come qualità del codice, build, release e pubblicazione del sito siano integrate in un unico flusso automatizzato.

7.14.1 Comandi principali

```
# Esecuzione test
sbt test

# Esecuzione test con coverage
sbt clean coverage test coverageReport

# Compilazione
sbt clean compile

# Creazione jar CLI
sbt clean assembly

# Generazione documentazione
sbt doc
```

7.14.2 Uso del jar

```
java -jar declslides.jar --input slides.sc --format html --output deck.html
java -jar declslides.jar --input slides.sc --format markdown --output deck.md
java -jar declslides.jar --input slides.sc --format text --output deck.txt
```

7.14.3 Deploy del sito

Il sito statico del progetto viene costruito come artefatto e poi pubblicato automaticamente tramite GitHub Pages. Questa scelta trasforma la documentazione del progetto in una parte viva del processo di rilascio.

7.15 Documentazione

La documentazione è stata trattata come parte integrante del progetto, e non come attività residuale. Questo si riflette in più livelli:

- documentazione inline del codice tramite Scaladoc;
- README e materiali di utilizzo;
- sito web di presentazione;
- diagrammi architetturali;
- guide operative per build, test ed esecuzione.

8. Sviluppo Iterativo e Sprint

Pur non essendo stato formalizzato come Scrum rigoroso, il progetto è stato chiaramente sviluppato per fasi incrementali.

Sprint/Fase	Obiettivo	Risultato	Criticità	Valore prodotto
Fase 1	modellazione dominio e DSL base	presentazione, slide, testo, bullet, codice	definizione dei vincoli	base concettuale del sistema
Fase 2	renderer HTML e testo	output multipli	separazione fra modello e vista	validazione del design multi-renderer
Fase 3	CLI e orchestrazione	esecuzione da terminale	gestione errori e bootstrap	usabilità reale del progetto
Fase 4	refactor architetturali	parser, factory, utility, error handling	evitare opacità	maggiore manutenibilità
Fase 5	immagini, footer, Markdown	nuove estensioni funzionali	mantenere coerenza del dominio	dimostrazione di estendibilità
Fase 6	Documentazione	pubblicazione su Pages	coordinamento con release	comunicazione e accessibilità del progetto

8.1 Tabella di dettaglio

Id	Item	Stima (h)	Effettivo (h)	S1	S2	S3
1	Setup repository e convenzioni di versionamento	2	3	3	0	0
2	Setup sbt, dipendenze e framework base	3	5	5	0	0
3	Setup CI multi-OS con GitHub Actions	3	5	4	1	0
4	Setup qualità: formatter, linter/ scalafix, coverage e Codecov	3	5	3	2	0
5	Modellazione del dominio e regole di validazione	5	8	5	3	0
6	DSL base e modello di composizione delle slide	6	9	4	5	0
7	Application layer, bootstrap ed esecuzione con Scala CLI	5	8	0	6	2
8	CLI, parsing argomenti, gestione errori e UX da terminale	4	6	0	5	1
9	Renderer testuale	3	4	0	3	1
10	Renderer HTML e navigazione della presentazione	5	9	0	4	5
11	Renderer Markdown	3	4	0	0	4
12	Supporto a immagini e footer nel DSL e nei renderer	5	7	0	0	5
13	Refactor per estendibilità, pulizia package e separazione responsabilità	4	5	0	0	3
14	Estensione suite di test, TDD hardening e regressioni	4	6	0	0	0

Id	Item	Stima (h)	Effettivo (h)	S1	S2	S3
15	Documentazione tecnica, sito di presentazione, Pages deploy e release automation	5	6	0	0	0
TOT		**60**	**90**	**24**	**29**	**21**

8.2 Tabella di riepilogo

Fase	Attività principali	Stima (h)	Effettivo (h)	Scostamento
Analisi	comprensione del dominio, definizione obiettivi, requisiti iniziali	8	12	+4
Progettazione	architettura, organizzazione package, modellazione DSL e dominio	12	18	+6
Setup tecnico	repository, sbt, framework, CI, coverage, formatter, linter	10	16	+6
Implementazione core	DSL, dominio, application layer, CLI	14	20	+6
Rendering	HTML, text, Markdown, immagini, footer	8	12	+4
Testing e refactor	TDD, regressioni, pulizia architetturale, manutenzione	4	7	+3
Documentazione e presentazione	Scaladoc, sito, immagini, relazione, deploy Pages	4	5	+1
TOT		**60**	**90**	**+30**

9. Problemi incontrati e soluzioni

9.1 Problema 1: rischio di accoppiamento tra DSL e dominio

All'inizio era facile far confluire troppe responsabilità nel DSL. La soluzione adottata è stata mantenere il DSL come layer di composizione e delegare la validazione al dominio. Questo ha reso più chiaro il ruolo di ciascun componente.

9.2 Problema 2: crescita della CLI e opacità del flusso

La CLI inizialmente rischiava di diventare un punto di concentrazione eccessiva. Il refactor ha portato all'estrazione di parser, config, errori, usage e programma CLI, rendendo il flusso più leggibile e più testabile.

9.3 Problema 3: gestione degli asset e script nel renderer HTML

L'integrazione di uno script di navigazione e di risorse statiche ha richiesto attenzione. Si è scelto di rendere più esplicita la gestione delle resource e di evitare fallback silenziosi che avrebbero nascosto errori reali.

9.4 Problema 4: mantenere estendibilità durante l'aggiunta di nuove feature

L'introduzione di immagini, footer e renderer Markdown avrebbe potuto generare modifiche troppo invasive. Il fatto che le estensioni siano state integrate con modifiche mirate mostra che l'architettura era già predisposta a crescere.

9.5 Problema 5: equilibrio fra pulizia architetturale e pragmatismo

Non tutte le scelte "perfette" erano giustificate dal contesto. In alcuni casi si è preferita una soluzione semplice ma pulita, evitando di introdurre complessità eccessiva per problemi ancora piccoli.

10. Sintesi

DeclSlides è un sistema software progettato per affrontare un problema molto concreto: la difficoltà di costruire presentazioni in modo riproducibile, tipizzato, controllabile e facilmente estendibile quando si lavora in un contesto tecnico. Nella pratica, gran parte degli strumenti di presentazione più diffusi è fortemente orientata all'editing visuale, ma meno adatta a chi desidera trattare una presentazione come artefatto software, versionabile, testabile e integrabile in una pipeline di build.

Il progetto introduce quindi un DSL dichiarativo in Scala che permette di descrivere una presentazione come codice strutturato. L'utente definisce il titolo del deck, il tema, l'eventuale footer, l'insieme delle slide, il layout di ciascuna slide e i singoli elementi di contenuto, come paragrafi, bullet list, blocchi di codice, spaziatori e immagini. Il risultato di questa descrizione non resta confinato al solo modello dati: il sistema è in grado di validare il contenuto e di renderizzarlo in diversi formati, con un approccio coerente e unificato.

Dal punto di vista architetturale, il progetto è stato costruito con una separazione netta fra dominio, DSL, application layer, rendering e interfaccia CLI. Il dominio definisce le regole e i vincoli della presentazione; il DSL fornisce una sintassi leggibile e guidata; il livello applicativo orchestra validazione, bootstrap ed esecuzione; il rendering traduce una presentazione validata nei diversi output; la CLI rende il sistema utilizzabile da terminale attraverso uno script `.sc` e un `jar eseguibile`. Questa organizzazione migliora leggibilità, manutenibilità, testabilità ed estendibilità.

Le tecnologie adottate sono state selezionate con criterio. `Scala 3` offre un linguaggio espressivo adatto a modellazione, immutabilità e DSL; `sbt` governa build e test; `Scala CLI` viene sfruttato per l'esecuzione degli script utente; `ScalaTags` supporta la generazione del markup HTML; `ScalaTest` garantisce una suite di test leggibile e coerente. Sul piano DevOps, `GitHub Actions` automatizza test multi-piattaforma, build, rilascio e pubblicazione del sito di presentazione del progetto.

Il valore del progetto non risiede solo nel prodotto finale, ma anche nel percorso ingegneristico seguito. DeclSlides dimostra come principi di progettazione del software, validazione esplicita, refactoring iterativo e TDD possano essere applicati con efficacia anche a un dominio apparentemente semplice come quello delle presentazioni. Il risultato è un sistema piccolo ma serio, con una chiara identità architetturale, un'attenzione concreta alla qualità del codice e una base solida per sviluppi futuri.

11. Retrospettiva

Guardando il progetto nel suo insieme, emergono diversi aspetti positivi. Il primo è la coerenza architetturale: il sistema non appare come un insieme di soluzioni casuali, ma come un piccolo ecosistema con ruoli ben definiti. Il secondo è la qualità del percorso di sviluppo: TDD, refactoring e modularizzazione hanno contribuito a costruire un codice leggibile e relativamente facile da estendere.

Ha funzionato bene anche il collegamento fra esigenze pratiche e obiettivi formativi. DeclSlides non è solo una dimostrazione teorica di DSL e renderer: è anche uno strumento realmente utilizzabile per generare presentazioni da script.

Le aree migliorabili riguardano soprattutto l'evoluzione verso una struttura più ampia: packaging della libreria, supporto a plugin, raffinamento del sito, e potenziale separazione più netta tra core e CLI in ottica distribuzione.

Dal punto di vista delle competenze, il progetto ha permesso di consolidare abilità in:

- modellazione di dominio;
- progettazione di DSL;
- testing in Scala;
- CI/CD;
- refactoring architetturale;
- documentazione tecnica tramite mkdocs;

12. Sviluppi futuri

Gli sviluppi futuri del progetto possono essere articolati in più direzioni.

12.1 Miglioramenti funzionali

- esportazione PDF;
- supporto a elementi avanzati come tabelle o citazioni;
- possibilità di cambiare font e stile della presentazione, definendone uno personale;
- possibilità di definire animazioni per ogni elemento della slide;
- pubblicazione tramite CI della presentazione su servizi di hosting.

12.2 Miglioramenti tecnici

- plugin renderer esterni;
- miglioramento del bootstrap e del runtime;
- eventuale passaggio a un modello asset più strutturato per HTML.

12.3 Limiti noti

Il renderer HTML è il formato più curato, mentre Markdown e testo semplice sono pensati soprattutto come output di ispezione.

In particolare, alcune informazioni visuali, come il layout centrato, non hanno ancora una rappresentazione forte in Markdown.

Una possibile evoluzione sarebbe introdurre metadata espliciti nel Markdown o distinguere tra renderer “di presentazione” e renderer “di documentazione”.

13. Conclusione

DeclSlides rappresenta un progetto software di dimensione contenuta ma di forte densità ingegneristica. Il problema affrontato, descrivere e generare presentazioni in modo dichiarativo, validato e multi-formato, è stato risolto con una soluzione coerente, tecnicamente solida e progressivamente estesa senza compromettere la struttura del sistema.

Gli obiettivi iniziali sono stati raggiunti: il progetto permette di definire una presentazione tramite DSL, validarla con regole esplicite, renderizzarla in più formati, eseguirla da CLI e presentarla tramite un sito dedicato. Ancora più importante, queste funzionalità non sono state costruite in modo monolitico o improvvisato, ma inserite in un’architettura chiara, con una forte attenzione alla separazione delle responsabilità, alla leggibilità del codice e alla qualità del processo di sviluppo.

Le principali scelte ingegneristiche, dominio tipizzato, error handling esplicito, layer distinti, registry dei renderer, refactoring progressivi, TDD, hanno dato al progetto una struttura che va oltre l’esercizio didattico. DeclSlides si presenta come un sistema piccolo, ma progettato seriamente, capace di mostrare non soltanto un risultato finale funzionante, ma anche un percorso di sviluppo consapevole.

Dal punto di vista formativo e professionale, il progetto ha permesso di consolidare competenze fondamentali di ingegneria del software: analisi, modellazione, design, test, automazione, documentazione e comunicazione tecnica. Per questo motivo, DeclSlides può essere considerato non solo un prodotto software riuscito, ma anche una dimostrazione concreta di metodo, rigore e maturità progettuale.