



Product Baseline - Allegato tecnico

Informazioni Documento

Distribuzione	Prof. Tullio Vardanega
	Prof. Riccardo Cardin
	Gruppo Graphite
Uso	Esterno
Recapito	graphite.swe@gmail.com



Indice

1	Introduzione	3
1.1	Scopo del documento	3
1.2	Scopo del prodotto	3
1.3	Riferimenti	3
2	Requisiti di sistema	5
3	Installazione ed esecuzione	6
4	Rapporto con il PoC	7
5	Architettura	8
5.1	Architettura generale del prodotto	8
5.2	Implementazione Model-View-ViewModel	11
5.3	View	11
5.4	Model	11
5.4.1	Contestualizzazione	11
5.4.2	Diagramma delle classi	12
5.4.3	Design pattern	12
5.4.3.1	Command	12
5.4.3.2	Builder	13
5.4.3.3	Facade	13
5.5	ViewModel	14
5.5.1	Contestualizzazione	14
5.5.2	Diagramma delle classi	14
5.5.3	Design pattern	15
5.5.3.1	Observer	15
5.6	Diagrammi di sequenza	15
5.7	Caricamento voice	15
5.8	Esecuzione utterance processor selezionati	16



5.9	Stampa grafo	17
6	Use case coperti	19
6.1	Tabella della copertura degli use case	19
6.2	Grafico della copertura degli use case	19
7	Requisiti soddisfatti	20
7.1	Tabella del soddisfacimento dei requisiti	20
7.2	Grafico del soddisfacimento dei requisiti	20
A	Model-View-ViewModel	21
A.1	Struttura del pattern	21
A.2	Vantaggi offerti dal pattern	22



1. Introduzione

1.1 Scopo del documento

Il documento ha la finalità di illustrare la *Product Baseline* (PB in breve) per l'applicazione "*DeSpeect: un'interfaccia grafica per Speect*", con particolare attenzione per lo stato attuale del prodotto, la sua architettura e la copertura di use case e requisiti funzionali obbligatori.

1.2 Scopo del prodotto

Lo scopo del progetto è la realizzazione di un'interfaccia grafica per *Speect_G* [Meraka Institute(2008-2013)], una libreria per la creazione di sistemi di sintesi vocale, che agevoli l'ispezione del suo stato interno durante il funzionamento e la scrittura di test per le sue funzionalità.

1.3 Riferimenti

Riferimenti normativi

- **Norme di Progetto v3.0.0:** documento *Norme di progetto v3.0.0*;
§2.2.5 "Progettazione";
§4.7.3 "Strumenti relativi allo sviluppo".

Riferimenti informativi

- **Analisi dei Requisiti v3.0.0:** documento *Analisi dei Requisiti v3.0.0*;
Definisce nel dettaglio use case e requisiti.
- **Documentazione Speect:**
<http://speect.sourceforge.net/contents.html>;



Documentazione ufficiale della libreria di *Text-To-Speech* di riferimento per il progetto.

- **Documentazione Qt:**

<http://doc.qt.io/>;

Documentazione ufficiale del framework utilizzato per lo sviluppo dell'interfaccia grafica.

- **Documentazione CMAKE:**

<https://cmake.org/documentation/>.

Documentazione ufficiale del framework utilizzato per la build del prodotto.



2. Requisiti di sistema

L'installazione ed esecuzione del software richiede i seguenti prerequisiti:

- Sistema operativo Unix / Unix-like (il software è stato testato solo per piattaforma Ubuntu 16.04 LTS)

<https://www.ubuntu.com/download/desktop>

- CMake (versione minima 2.8)

<https://cmake.org/download/>

- Compilatore ANSI C/ISO C90 GCC (versione minima 5.0)

<https://gcc.gnu.org/install/binaries.html>

- Qt 5.9.0

<https://www.qt.io/download>



3. Installazione ed esecuzione

Il codice relativo alla Product Baseline è reperibile da un apposito repository GitHub al seguente link:

<https://github.com/graphiteSWE/Despeect-ProductBaseline>

Per installare ed eseguire il software è necessario attenersi alla seguente procedura:

1. Clonare o scaricare la repository sulla propria macchina;
2. Entrare nella cartella scaricata ed eseguire lo script `build.sh`;
3. Eseguire da terminale il comando `cd DeSpeect/build/`;
4. Avviare l'eseguibile con il comando `./main`.

Tale procedura installerà la libreria Speect e genererà una build del software nella directory `DeSpeect/build`, nonché avvierà automaticamente un'esecuzione dello stesso. Ulteriori informazioni sono reperibili nel file `README.md` del repository.



4. Rapporto con il PoC

Precedentemente alla Product Baseline, è stato realizzato un *Proof of Concept* (PoC in breve) a dimostrazione della fattibilità del prodotto DeSpeect. Tale PoC è tuttora reperibile al seguente link:

<https://github.com/graphiteSWE/TB-PoC>

Segue una tabella che evidenzia le differenze tra le caratteristiche del PoC, contestualizzato nel suo scopo, e quelle della PB.

Tabella 4.1: Tabella di confronto tra PoC e PB

Proof of Concept	Product Baseline
Architettura abbozzata e sommaria, priva di design pattern	Architettura basata su MVVM e facente uso di vari design pattern
Implementazione di un'interfaccia grafica provvisoria e carente di elementi fondamentali per il soddisfacimento di molti requisiti obbligatori	Interfaccia grafica quasi completa predisposta all'implementazione della totalità dei requisiti obbligatori
Implementazione di poche funzionalità dimostrative (per esempio la stampa parziale del grafo)	Implementazione della maggior parte delle funzionalità richieste dai requisiti funzionali obbligatori
Modalità di installazione e configurazione macchinose	Modalità di installazione e configurazione semplificate



5. Architettura

5.1 Architettura generale del prodotto

L'architettura generale del prodotto segue il pattern Model-View-ViewModel. Questo pattern è basato su tre componenti principali:

- **Model:** un'implementazione del modello del dominio dei dati;
- **View:** la struttura, il layout e l'aspetto di ciò che l'utente visualizza a schermo;
- **ViewModel:** un'astrazione della view che espone proprietà pubbliche e comandi.

Esso permette tra le altre cose un totale disaccoppiamento tra logica di business e presentazione, informazioni più dettagliate a riguardo sono reperibili nell'appendice §A "Model-View-ViewModel" di questo documento. I seguenti diagrammi illustrano sinteticamente la struttura del software con livello di dettaglio crescente.

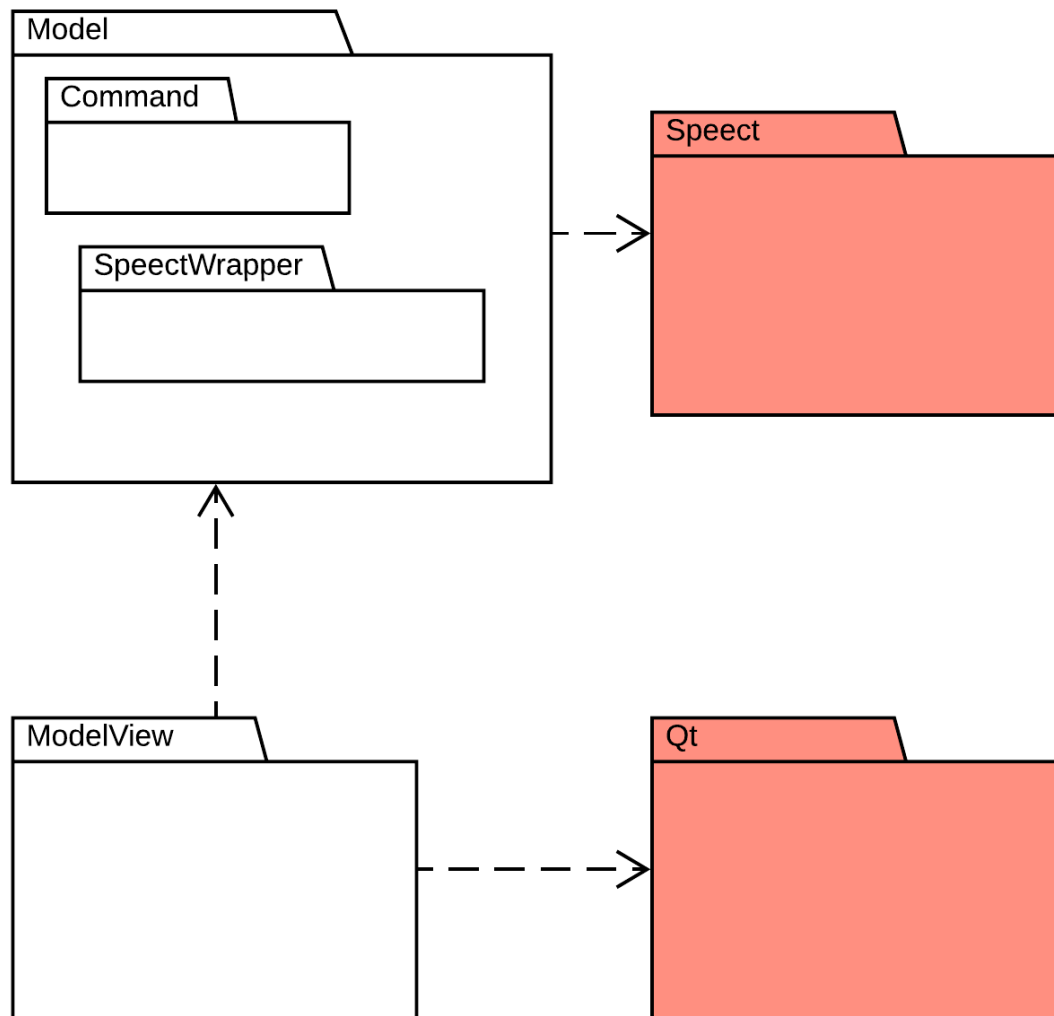


Figura 5.1: Struttura generale dell'architettura

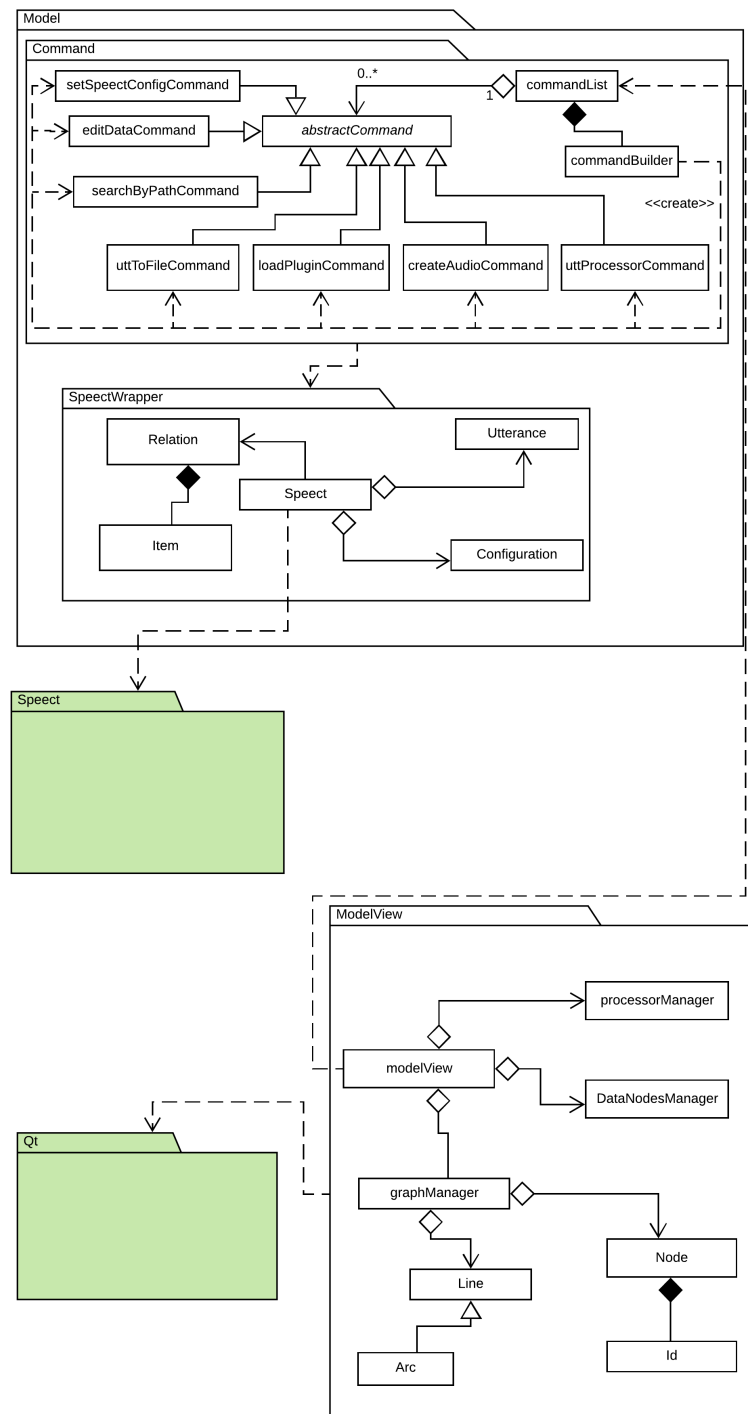


Figura 5.2: Struttura dei package



5.2 Implementazione Model-View-ViewModel

Vengono di seguito illustrate le implementazioni per le componenti del pattern Model-View-ViewModel in relazione all'architettura della Product Baseline. Per ogni componente, vengono illustrati:

- **Contestualizzazione:** spiegazione generale dell'architettura del componente.
- **Diagramma delle classi:** diagramma generale delle classi per il componente. Per motivi di spazio, il diagramma qui riportato è una versione semplificata priva dell'indicazione di tutti i metodi. Un link al diagramma completo viene riportato in questa sezione, ed è inoltre presente sul repository un'apposita directory `Diagrammi/Diagrammi delle classi/` contenente la totalità degli stessi;
- **Design Pattern:** una descrizione e contestualizzazione esaustiva dei design pattern impiegati all'interno del componente.

5.3 View

La View, conseguentemente all'uso del framework Qt, consiste di un file qml trasformato durante la compilazione in classi compatibili C++. Il comportamento della View è infatti gestito dal package ViewModel.

5.4 Model

5.4.1 Contestualizzazione

Nella progettazione del Model è emersa la necessità di interagire con la libreria Speect incapsulandone alcune funzionalità rilevanti. Per realizzare ciò il Model è stato diviso in due package corrispondenti all'implementazione dei design pattern Façade (SpeectWrapper), per quanto riguarda l'incapsulamento della libreria, e Command, per quanto riguarda la suddivisione delle funzionalità implementate in un'ottica di componibilità ed estendibilità.



```
classDiagram
    class Utterance {
        +std::string name
        +std::vector<std::string> words
    }
    class Configuration {
        +std::string path
        +std::string host
        +std::string port
    }
    class Item {
        +std::string name
        +std::string value
    }
    class Relation {
        +std::string key
        +std::string value
    }
    class Speect {
        +std::string name
        +std::vector<Utterance> utterances
        +Configuration configuration
        +std::vector<Item> items
        +std::vector<Relation> relations
    }
    class AbstractCommand {
        +execute(speectEngine : Speect*) : std::string
    }
    class SearchByPathCommand {
        +execute(speectEngine : Speect*) : std::string
    }
    class EditDataCommand {
        +execute(speectEngine : Speect*) : std::string
    }
    class SetSpeectConfigCommand {
        +execute(speectEngine : Speect*) : std::string
    }
    class LoadPluginCommand {
        +execute(speectEngine : Speect*) : std::string
    }
    class CreateAudioCommand {
        +execute(speectEngine : Speect*) : std::string
    }
    class UttToFileCommand {
        +execute(speectEngine : Speect*) : std::string
    }
    class UttProcessorCommand {
        +execute(speectEngine : Speect*) : std::string
    }
    class CommandList {
        +executeAll() : void
        +executeStep() : void
    }
    class CommandBuilder {
    }
    class SpeectWrapper {
    }
    class SpeectView {
    }

    Utterance "1" o-- "*" Speect
    Configuration "1" o-- "*" Speect
    Item "1" *-- "*" Speect
    Relation "1" *-- "*" Speect
    AbstractCommand <|-- SearchByPathCommand
    AbstractCommand <|-- EditDataCommand
    AbstractCommand <|-- SetSpeectConfigCommand
    AbstractCommand <|-- LoadPluginCommand
    AbstractCommand <|-- CreateAudioCommand
    AbstractCommand <|-- UttToFileCommand
    AbstractCommand <|-- UttProcessorCommand
    CommandList o-- "*" AbstractCommand
    CommandList o-- "*" Speect
    CommandBuilder *--> "1" CommandList : <<create>>
    SpeectWrapper --> Speect
    SpeectView --> Speect
```

The UML class diagram illustrates the Command pattern implementation for the Speect system. It features several classes and their relationships:

- SpeectWrapper**: A wrapper class that interacts with the **Speect** object.
- Speect**: The main entity, containing references to **Utterance**, **Configuration**, **Item**, and **Relation**. It has aggregation relationships with **Utterance** and **Configuration**, and composition relationships with **Item** and **Relation**.
- Utterance**: Contains `+std::string name` and `+std::vector<std::string> words`.
- Configuration**: Contains `+std::string path`, `+std::string host`, and `+std::string port`.
- Item**: Contains `+std::string name` and `+std::string value`.
- Relation**: Contains `+std::string key` and `+std::string value`.
- AbstractCommand**: An abstract base class for commands, defining `+ execute(speectEngine : Speect*) : std::string`.
- SearchByPathCommand**, **EditDataCommand**, **SetSpeectConfigCommand**, **LoadPluginCommand**, **CreateAudioCommand**, **UttToFileCommand**, and **UttProcessorCommand**: Concrete command classes implementing the `execute` method.
- CommandList**: Manages a collection of **AbstractCommand** objects and provides methods `+ executeAll () : void` and `+ executeStep () : void`. It has an aggregation relationship with **AbstractCommand** and a reference to **Speect**.
- CommandBuilder**: Responsible for creating **CommandList** objects, indicated by the `<<create>>` note.
- Si interfaccia con ViewModel**: A note indicating that the **CommandBuilder** class interfaces with a **ViewModel**.

5.4.3 Design pattern

Permette la suddivisione delle funzionalità implementate in un'ottica di componibilità ed estendibilità. I comandi concreti sono aggregati in una lista (**CommandList**) che si interfaccia con la componente ViewModel. Tali co-



mandi interagiscono a loro volta con il package SpeectWrapper per ottenere i dati da elaborare dalla libreria.

5.4.3.2 Builder

Questo design pattern separa la costruzione di un oggetto complesso dalla sua rappresentazione cosicché il processo di costruzione stesso possa creare diverse rappresentazioni. Contestualizzato nel sistema della PB, esso si interfaccia con il ViewModel per la configurazione e costruzione della `CommandList`.

5.4.3.3 Façade

Il design pattern Façade permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e molto diverse tra loro, nonché a blocchi di codice complessi. In questo contesto, il package SpeectWrapper incapsula la libreria Speect rendendola accessibile tramite omonima classe proprietaria.



5.5 ViewModel

5.5.1 Contestualizzazione

Il package ViewModel funge da tramite tra Model e View, prelevando i dati dal primo per aggiornare il secondo. Per quanto riguarda la stampa del grafo, la classe **GraphManager** si occupa di aggiornarne la presentazione interfacciandosi con le librerie Qt e con le classi **Line Arc** e **Node**.

5.5.2 Diagramma delle classi

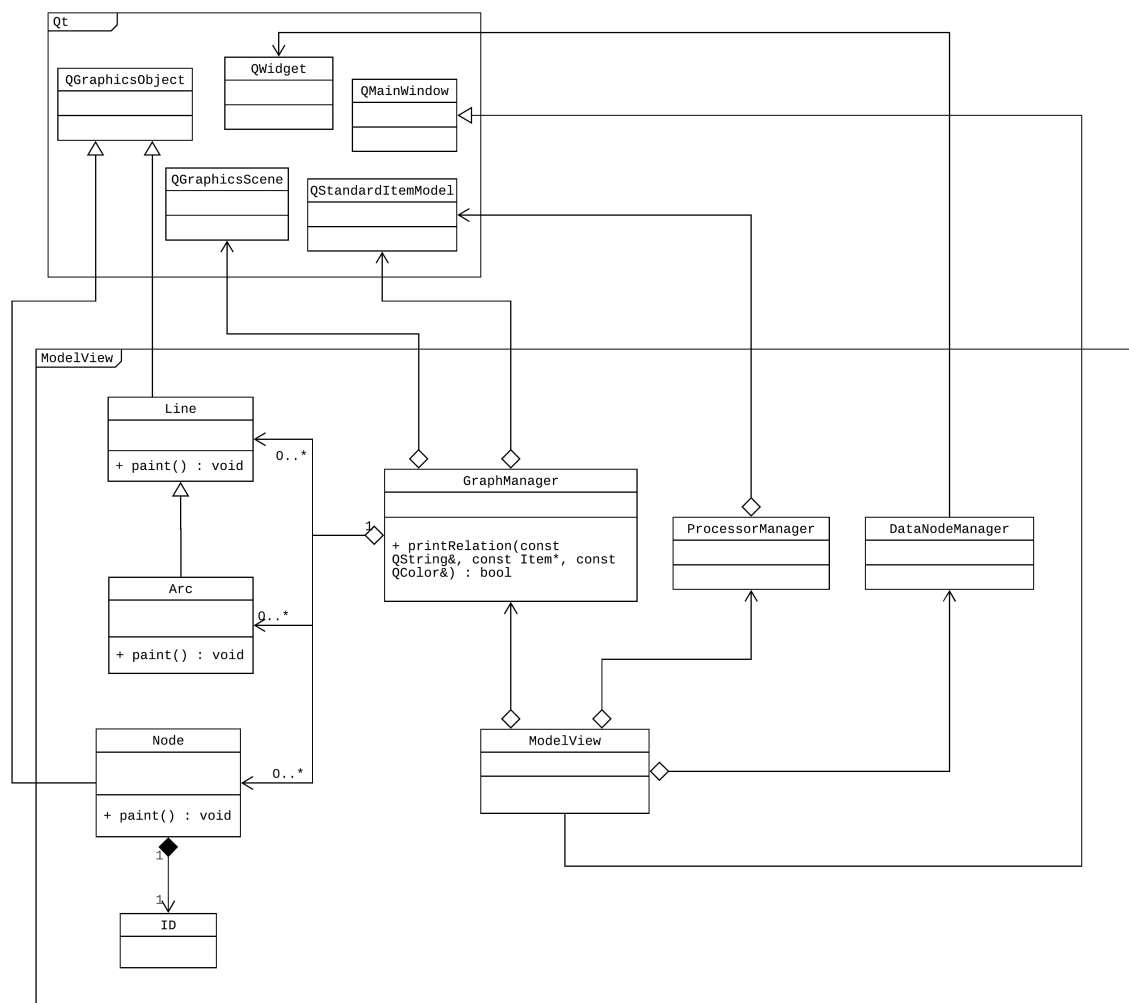


Figura 5.4: Struttura dei package



5.5.3 Design pattern

5.5.3.1 Observer

Il framework Qt, attraverso il sistema di signal e slot, implementa tale design pattern. Esso permette di reagire efficientemente ad un cambiamento dell'interfaccia grafica, chiedendo se necessario l'aggiornamento dei dati del Model attraverso il ViewModel. Il meccanismo di signal e slot è implementato dalle classi pertinenti all'interno del package.

5.6 Diagrammi di sequenza

Vengono qui presentati i diagrammi di sequenza per alcune richieste notevoli. Si noti che tali diagrammi sono disponibili all'interno del repository nella directory `Diagrammi/Diagrammi di sequenza/`.

5.7 Caricamento voice

Il seguente diagramma di sequenza rappresenta il processo di caricamento e configurazione di una voice all'interno del sistema.

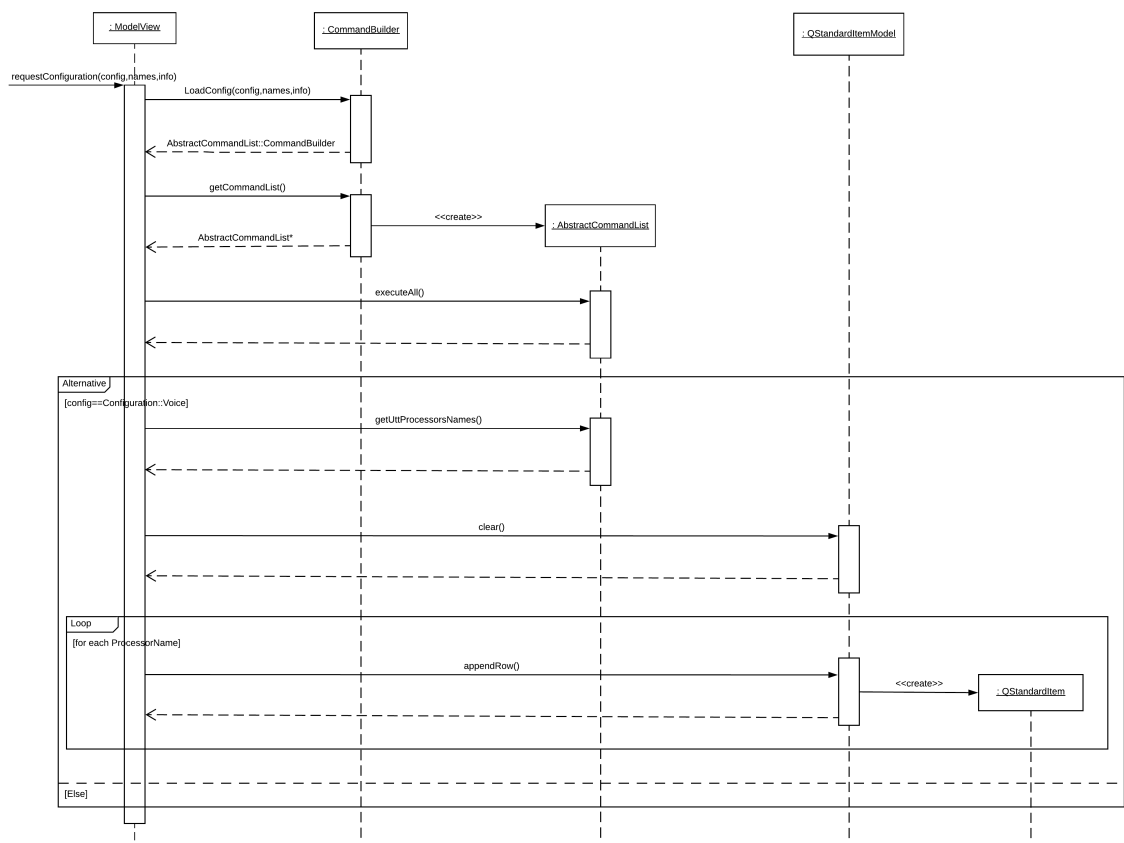
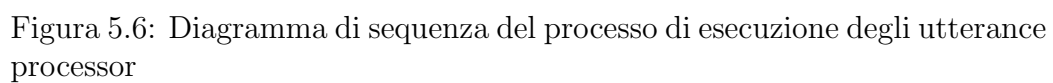


Figura 5.5: Diagramma di sequenza del processo di caricamento della voce

5.8 Esecuzione utterance processor selezionati

Il seguente diagramma di sequenza rappresenta il processo di esecuzione della lista di comandi selezionati dall'utente mediante l'interfaccia grafica.



Il seguente diagramma di sequenza rappresenta il processo di stampa di un grafo.

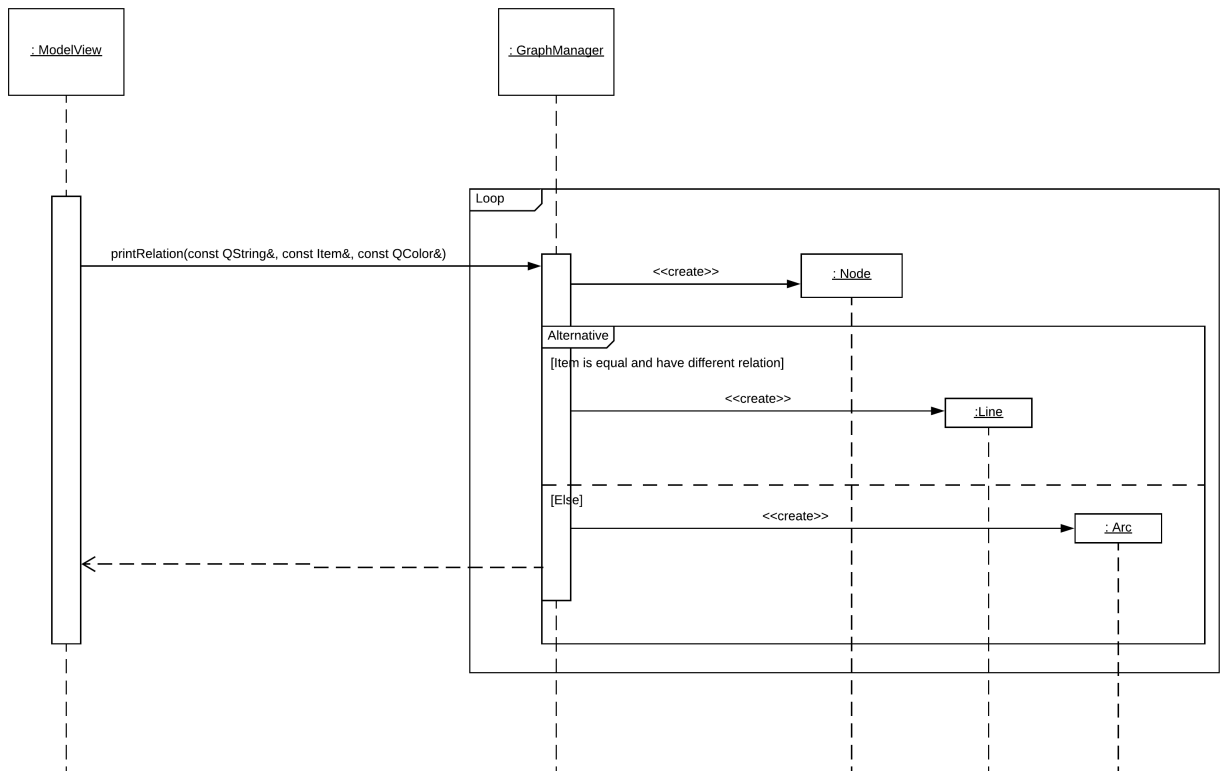


Figura 5.7: Diagramma di sequenza del processo di stampa del grafo



6. Use case coperti

6.1 Tabella della copertura degli use case

6.2 Grafico della copertura degli use case



7. Requisiti soddisfatti

7.1 Tabella del soddisfacimento dei requisiti

7.2 Grafico del soddisfacimento dei requisiti

A. Model-View-ViewModel

A.1 Struttura del pattern

Il design pattern architetturale *Model-View-ViewModel* (MVVM in breve) facilita la separazione dell'interfaccia grafica, che si tratti di linguaggio di markup o codice GUI, dallo sviluppo della logica di business o della logica di back-end, ovvero dal modello dei dati. Il *view model* di MVVM è un convertitore di valori, nel senso che il view model è responsabile dell'esposizione (conversione) degli oggetti dati dal modello così da renderli facilmente gestibili e presentabili. In quest'ottica, la view è più un modello che una vista e gestisce la maggior parte se non tutta la logica di visualizzazione della vista. Il pattern è riassunto dal seguente schema ed i suoi componenti principali sono di seguito approfonditi.

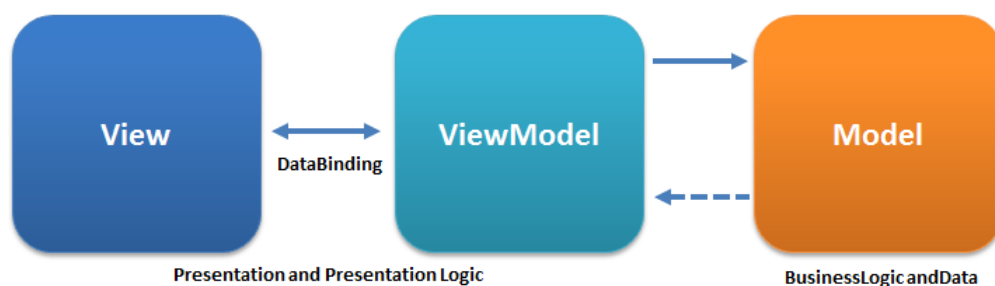


Figura A.1: Diagramma generale dell'architettura MVVM

I tre componenti principali dell'architettura sono i seguenti:

- **Model:** il *Model* (o modello) è un'implementazione del modello di dominio dell'applicazione ed include un modello dei dati affiancato alla logica di business e di validazione;



- **View:** la *View* (o vista) è responsabile della definizione della struttura, del layout e dell'aspetto di ciò che l'utente visualizza su schermo. Idealmente, la vista è definita puramente con linguaggio di markup o generico codice GUI che non contiene la logica di business;
- **ViewModel:** la *ViewModel* (o modello di presentazione) funge da intermediario tra la vista e il modello ed è responsabile della gestione della logica di visualizzazione. In genere, il ViewModel interagisce con il modello richiamandone i metodi delle classi: esso fornisce quindi dati dal modello in una forma facilmente utilizzabile dalla View. Il ViewModel recupera i dati dal modello, rendendoli disponibili alla View, e può riformattarli in un modo che renda più semplice la gestione della vista. Esso fornisce anche l'implementazione dei comandi che un utente dell'applicazione avvia nella vista (ad esempio, quando un utente clicca un pulsante nell'interfaccia grafica, tale azione può attivare un comando nel ViewModel) e può essere responsabile della definizione delle modifiche dello stato logico che influiscono su alcuni aspetti della visualizzazione nella vista, ad esempio l'indicazione che alcune operazioni sono in sospeso.

A.2 Vantaggi offerti dal pattern

Il MVVM offre i seguenti vantaggi:

- Durante il processo di sviluppo, i programmatori e i designer possono lavorare in modo indipendente e simultaneamente sui loro componenti. Quest'ultimi possono concentrarsi sulla vista e, utilizzando appositi strumenti, generare facilmente dati di esempio con cui lavorare, mentre i programmatori possono lavorare sul modello di presentazione e sui componenti del modello;
- Gli sviluppatori possono creare test unitari per il ViewModel e per il Model senza utilizzare la View;
- È facile riprogettare l'interfaccia grafica dell'applicazione senza toccare il resto del codice, una nuova versione della vista dovrebbe poter funzionare con il modello di presentazione esistente;
- Se esiste un'implementazione del modello che incapsula la logica di business, potrebbe essere difficile o rischioso cambiarla. In questo scenario, il ViewModel funge da adattatore per le classi del Model e consente di evitare modifiche importanti al codice dello stesso.