Specifica Tecnica

Gruppo SWEet BIT — Progetto SWEDesigner

Informazioni sul documento

Versione	1.0.0
Redazione	Santimaria Davide
	Massignan Fabio
Verifica	Massignan Fabio
	Bodian Malick
Approvazione	Pilò Salvatore
Uso	Esterno
Distribuzione	Prof. Tullio Vardanega
	Prof. Riccardo Cardin
	Gruppo SWEet BIT
	Zucchetti S.p.A.

Descrizione

Questo documento descrive la specifica tecnica e l'architettura del prodotto sviluppato dal gruppo SWEet BIT per la realizzazione del progetto SWEDesigner.

Versioni del documento

Versione	Data	Persone coinvolte	Descrizione
1.y.z	2017/??/??	Pilò Salvatore	Approvazione Documento
1.y.z	2017/??/??	Massignan Fabio	Verifica Documento
1.0.3	2017/05/02	NOME	Stesura sezione Descrizione architettura
1.0.2	2017/05/02	NOME	Stesura sezione Tecnologie utilizzate
1.0.1	2017/05/02	NOME	Stesura sezione Introduzione
1.0.0	2017/05/02	Santimaria Davide	Creazione struttura documento

Indice

Elenco delle figure

1 Introduzione

1.1 Scopo del documento

Questo documento ha come scopo quello di definire la *progettazione ad alto livello*$_G$ per il prodotto. Verrà presentata la strttura generale secondo la quale saranno organizzate le varie componenti software e i *Design Pattern*$_G$ utilizzati nella creazione del prodotto SWEDesigner. Verrà dettagliato il tracciamento tra le componenti software individuate ed i requisiti.

1.2 Scopo del prodotto

Lo scopo del progetto è la realizzazone di una *Web App*$_G$ che fornisca all'*Utente*$_G$ un *UML*$_G$ *Designer*$_G$ con il quale riuscire a disegnare correttamente *Diagrammi*$_G$ delle *Classi*$_G$ e descrivere il comportamento dei *Metodi*$_G$ interni alle stesse attraverso l'utilizzo di *Diagrammi*$_G$ delle attività. La *Web App*$_G$ permetterà all'*Utente*$_G$ di generare *Codice*$_G$ *Java*$_G$ dall'insieme dei *diagrammi classi*$_G$ e dei rispettivi *metodi*$_G$.

1.3 Glossario

Con lo scopo di evitare ambiguità di linguaggio e di massimizzare la comprensione dei documenti, il gruppo ha steso un documento interno che è il *Glossario v*1.2.0. In esso saranno definiti, in modo chiaro e conciso i termini che possono causare ambiguità o incomprensione del testo.

1.4 Riferimenti

1.4.1 Normativi

- **Capitolato d'Appalto C6: SWEDesigner**
  http://www.math.unipd.it/~tullio/IS-1/2015/Progetto/C6p.pdf;
- **Norme di Progetto:** *Norme di Progetto v1.2.0.*
- **Analisi dei Requisiti:** *Analisi dei Requisiti v1.2.0.*

1.4.2 Informativi

- **Slide dell'insegnamento Ingegneria del Software modulo A:** http://www.math.unipd.it/~tullio/IS-1/2016/.
  - **Slides del corso di Ingegneria del Software mod. A: *Diagrammi delle classi*$_G$**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E03.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: Diagrammi dei package**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E04.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: Diagrammi di sequenza**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E05.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: Diagrammi di attività**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E06.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: *Design pattern*$_G$ strutturali: Decorator, Proxy, Facade, Adapter**:http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E07.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: *Design pattern*$_G$ creazionali: Singleton, Builder, Abstract Factory**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E08.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: *Design pattern*$_G$ comportamentali: Observer, Template Method, Command, Strategy, Iterator**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E09.pdf;
- **Design Patterns - E. Gamma, R. Helm, R. Johnson, J. Vlissides (Pearson Education, Addison-Wesley, 1995;**;
- ***Node.js*$_G$**: https://nodejs.org/dist/latest-v5.x/docs/api/;
- **MongoDB**: https://docs.mongodb.org/manual/;
- **HTML5**: http://www.w3schools.com/html/html5_intro.asp;
- **CSS3**: http://www.w3schools.com/css/css3_intro.asp;
- **ExpressJS**: http://expressjs.com/en/4x/api.html.
- **Mustache**: http://mustache.github.io/.

2 Tecnologie utilizzate

L'architettura è stata progettata utilizzando lo stack di ***MEAN***$_G$ (http://mean.io/),il quale comprende 4 tecnologie, alcune delle quali espressamente richieste nel *capitolato*$_G$ d'appalto. Vengono di seguito elencate e descritte le principali tecnologie impiegate comprese in ***MEAN***$_G$ e le motivazioni del loro utilizzo:

- **Node.js**: piattaforma per il *back-end*$_G$;
- **Express.js**: *framework*$_G$ per la realizzazione dell'applicazione web in *Node.js*$_G$;
- **MongoDB**: *database*$_G$ di tipo *NoSQL*$_G$ per la parte di recupero e salvataggio dei dati;
- **Mongoose**: *libreria*$_G$ per interfacciarsi con il driver di **MongoDB**;
- **Angular.js**: *framework*$_G$ *JavaScript*$_G$ la realizzazione del *front-end*$_G$.

2.1 Server

2.1.1 Node.js

Node.js è una *piattaforma*$_G$ software costruita sul motore *JavaScript*$_G$ di *Chrome*$_G$ che permette di realizzare facilmente applicazioni di rete scalabili e veloci. *Node.js*$_G$ utilizza *JavaScript*$_G$ come linguaggio di programmazione, e grazie al suo modello *event-driven*$_G$ con chiamate di input/output non bloccanti risulta essere leggero e e ciente.

2.1.1.1 Vantaggi

- **Approccio asincrono**: *Node.js*$_G$ permette di accedere alle risorse del sistema operativo in modalità *event-driven*$_G$ e non sfruttando il classico modello basato su processi concorrenti utilizzato dai classici web *server*$_G$. Ciò garantisce una maggiore efficienza in termini di prestazioni, poiché durante le attese il runtime può gestire qualcos'altro in maniera asincrona;
- **Architettura modulare**: Lavorando con *Node.js*$_G$ è molto facile organizzare il lavoro in librerie, importare i *moduli*$_G$ e combinarli fra loro.Questo è reso molto comodo attraverso il *node package manager*$_G$ (**npm**) attraverso il quale lo sviluppatore può contribuire e accedere ai *package*$_G$ messi a disposizione dalla community.

2.1.1.2 Svantaggi

- **Supporto incompleto alle feature di ES6**: Molte delle feature di ES6 non sono supportate in Node nella versione 4.4 scelta come versione di riferimento per lo sviluppo del progetto.

2.1.2 Mustache

Mustache è un web *template*$_G$ che permette di espandere tags all'interno di un *template*$_G$, racchiusi da 2 parentesi graffe,usando valori forniti da oggetti.

2.1.2.1 Vantaggi

- Si presenta come uno strumento dalle caretteristiche semplici, (*Logic-less templates*) in quanto rimuove i controlli di flusso if ed i cicli for, usa i solo tag di vario tipo per sostituirli;
- è usato principalmente per *Web-App*$_G$.

2.1.2.2 Svantaggi

- La lettura dei file potrebbe risultare difficoltosa a causa del numero elevato di parentesi per ogni tag, ad esempio ignorare una variabile la sintassi richiede 3 parentesi graffe.

2.2 Client

2.2.1 Express.js

Express.js è un *framework*$_G$ minimale per creare *Web App*$_G$ con *Node.js*$_G$. Richiede *moduli*$_G$ Node di terze parti per applicazioni che prevedono l'interazione con le *database*$_G$. È stato utilizzato il *framework*$_G$ *Express.js*$_G$ per supportare lo sviluppo dell'application *server*$_G$ grazie alle utili e robuste caratteristiche da esso offerte, le quali sono pensate per non oscurare le funzionalità fornite da *Node.js*$_G$ aprendo così le porte all'utilizzo di moduli per *Node.js*$_G$ atti a supportare specifiche funzionalità.

2.2.1.1 Vantaggi

- **Minimale**: si basa su *Node.js*$_G$ e permette di estenderlo a seconda dei bisogni dell'applicazione;
- **Documentazione**: esaustiva e completa;
- **Apprendimento**: facile da imparare.

2.2.1.2 Svantaggi

- **Integrazione**: richiede di integrare *moduli*$_G$ diversi per comporre l'applicazione finale. Altri *framework*$_G$ permettono di definire *API*$_G$ (Application Programming Interface) *REST*$_G$ (REpresentational State Transfer) in modo semplice, ma vincolano maggiormente nelle scelte progettuali.

2.2.2 MongoDB

MongoDB$_G$ è un *database*$_G$ *NoSQL*$_G$ *open source*$_G$ scalabile e altamente performante di tipo document-oriented, in cui i dati sono archiviati sotto forma di documenti in stile *JSON*$_G$ con schemi dinamici, secondo una struttura semplice e potente.

2.2.2.1 Vantaggi

- **Alte performance**: non ci sono join che possono rallentare le operazioni di lettura o scrittura. L'indicizzazione include gli indici di chiave anche sui documenti innestati e sugli array, permettendo una rapida interrogazione al *database*$_G$;
- **Affidabilità**: alto meccanismo di replicazione su server;
- **Schemaless**: non esiste nessuno *schema*$_G$, è più flessibile e può essere facilmente traspostoin un modello ad oggetti;
- Permette di definire query complesse utilizzando un linguaggio che non è *SQL*$_G$;
- Permette di processare parallelamente i dati (*Map-Reduce*$_G$);
- Tipi di dato più flessibili.

2.2.2.2 Svantaggi

- **Flessibilità**: per i tipi di dato. Sebbene questo possa essere visto come vantaggio, è opinione del team che un'eccessiva flessibilità possa portare più problemi che benefici: allo scopo di aggiungere rigidità è stato infatti scelto, come verrà descritto in seguito, Mongoose, che introduce una costruzione a schemi per le collections di *MongoDB*$_G$ e quindi vincola i documenti inseriti ad avere una struttura uniforme;
- **Nessun supporto per le transazioni**: sono supportate alcune operazioni atomiche, ma a livello di documento;
- **Nessun *join***$_G$: va simulato via codice attraverso query multiple;
- **Problemi di concorrenza**: per le operazioni di scrittura viene creato un lock sull'intero database. Questo lock blocca anche le operazioni di lettura.

2.2.3 Mongoose

Mongoose è una *libreria*$_G$ per interfacciarsi a *MongoDB*$_G$ che permette di definire degli schemi per modellare i dati del *database*$_G$, imponendo una certa struttura per la creazione di nuovi Document . Inoltre fornisce molti strumenti utili per la validazione dei dati, per la definizione di query e per il cast dei tipi predefiniti. Per interfacciare l'application *server*$_G$ con *MongoDB*$_G$ sono disponibili diversi progetti *open source*$_G$. Per questo progetto è stato scelto di utilizzare *Mongoose.js*$_G$, attualmente il più di uso.

2.2.3.1 Vantaggi

- **Diffusione**: è la libreria più diffusa per interfacciarsi con *MongoDB*$_G$;
- **Funzionalità aggiuntive**: permette di definire strumenti per la validazione dei dati e per il cast dei tipi;
- **Permette di eseguire dei *join*$_G$ tra collections**: Sebbene non sia previsto da *MongoDB*$_G$, *mongoose*$_G$ prevede la funzione populate per imitare la funzione di *join*$_G$ in modo completamente trasparente per l'utilizzatore;
- **Rapido ed intuitivo**: La strutturazione dei dati con questa libreria è rapida ed inutitiva, ciò dovuto anche dalla sintassi dichiarativa della libreria stessa.

2.2.3.2 Svantaggi

- **Schema-based**: è basato sulla creazione di una forte schematizzazione per i documenti, e questo limita l'estrema flessibilità di *MongoDB*$_G$.

2.2.4 mxGrafh

mxGraph è una libreria *JavaScript*$_G$ di *diagrammi*$_G$ che consente di creare rapidamente applicazioni di grafici interattive e grafici che vengono eseguiti in modo nativo su tutti i browser principali.

2.2.4.1 Vantaggi

- **Non sono necessari altri plug-in**: Ciò elimina i plug-in di dipendenza dai fornitori;
- ***Open-source*$_G$**:Le tecnologie coinvolte sono libere e ci sono molte implementazioni aperte, nessun fornitore può rimuovere un prodotto o una tecnologia che lascia in pratica la tua applicazione inoperabile;

- **Le tecnologie sono standardizzate**: L'applicazione è distribuibile al numero massimo di utenti del browser senza bisogno di ulteriori configurazioni o installazione sul computer *client*$_G$. Gli ambienti aziendali di grandi dimensioni spesso non amano consentire agli individui di installare plug-in del *browser*$_G$ e non amano cambiare la build standard creata su tutte le macchine.

2.2.4.2 Svantaggi

- **Aumento rapido di celle**: Poiché il numero di celle visibili sullo schermo degli utenti aumenta di centinaia, la valutazione rallenta oltre i limiti accettabili sulla maggior parte dei *browser*$_G$. Nella teoria della gestione delle informazioni, visualizzare diverse centinaia di celle è generalmente sbagliato, in quanto l'utente non può interpretare i dati.

2.2.5 HTML5

È un linguaggio di markup per la strutturazione delle pagine web, pubblicato come W3C Recommendation da ottobre 2014. L'uso di HTML5 rispetto a XHTML (e**X**tensible **H**yper**T**ext **M**arkup **L**anguage) è stato deciso all'unanimità dal gruppo.

2.2.5.1 Vantaggi

- Raccomandazione W3C;
- reazione di pagine interattive: soprattutto se usato insieme a CSS.

2.2.5.2 Svantaggi

- Supporto: non tutti i browser lo supportano allo stesso modo, e non tutte le caratteristiche definite sono ancora completamente supportate.

2.2.6 CSS3

È un linguaggio utilizzato per definire la formattazione di documenti HTML. Le regole per la composizione di un foglio di stile CSS sono definite dal W3C a partire dal 1996. Inoltre permette di separare i contenuti delle pagine HTML dalla loro formattazione, assicurando una maggiore manutenibilità e riutilizzo.

2.2.6.1 Vantaggi

- Separazione tra contenuto e presentazione;
- Raccomandato da W3C.

3 Descrizione architettura

3.1 Metodo e formalismo di specifica

Le scelte architetturali per lo sviluppo di SWEDesigner sono state fortemente influenzate dallo stack tecnologico utilizzato.

Nell'esposizione dell'architettura dell'applicazione si procederà con un approccio *top-down*$_G$, descrivendo l'architettura iniziando dal generale ed andando al particolare; si è partiti suddividendo il sistema in *front-end*$_G$ e *back-end*$_G$, definendo l'interfaccia di comunicazione, scegliendo di seguire in ciascuno l'organizzazione suggeritaci dai *framework*$_G$.

La descrizione dell'architettura di SWEDesigner è suddivisa in quattro sezioni:

- §3.2: illustra gli aspetti generali dell'architettura del software;
- §3.3: descrive il protocollo che lega le due interfacce tra *Client*$_G$ e *Server*$_G$; che descrive l'architettura del front end dell'applicazione;
- §3.4: descrive l'architettura del *back-end*$_G$ dell'applicazione;
- §3.5: descrive l'architettura del *front-end*$_G$ dell'applicazione.

Per descrivere in maniera formale l'architettura verranno impiegati lo standard *UML*$_G$ 2.0 per i diagrammi dei *package*$_G$ e delle classi e lo standard *UML*$_G$ 2.5 per i *diagrammi di attività*$_G$ e sequenza.

I diagrammi delle *classi*$_G$ che permettono di mostrare l'architettura generale del sistema vengono affiancati anche dai diagrammi di sequenza e attività, che permettono di definire le interazioni tra le componenti, senza preoccuparsi della loro classificazione. In questo modo è possibile esprimere alcuni meccanismi tipici di un'applicazione *REST*$_G$-like, come il modo in cui agiscono i *middleware*$_G$.

3.2 Architettura generale

L'architettura del progetto si divide in una componente *Client*$_G$, rappresentata da un'applicazione *front-end*$_G$ accessibile da un browser, e in una componente Web*Server*$_G$, nella quale risiede il *back-end*$_G$ che gestisce le richieste di generazione del *codice*$_G$.
L'architettura generale di SWEDesigner si divide in 3 macrocomponenti:

- ***Client***$_G$;
- ***Server***$_G$ ***REST***$_G$;
- ***Database***$_G$.

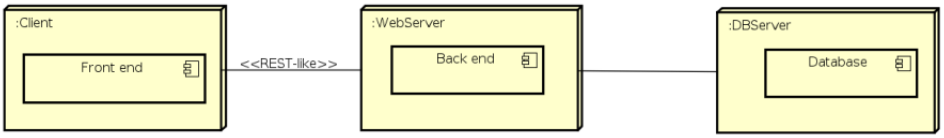

Figura 1: Diagramma di *deployment*$_G$ per l'architettura

L'architettura proposta segue il *Design Pattern*$_G$ MVC. In particolare i ruoli di Model e Controller verranno implementati a livello di *server*$_G$, mentre il ruolo di View viene affidato al *front-end*$_G$. L'interfaccia tra le due componenti verrà gestita grazie ad un set di *API*$_G$ disposto dal *server*$_G$ *REST*$_G$; il *Database*$_G$ serve per garantire la persistenza del programma generato: ogni *utente*$_G$ autenticato può salvare i propri progetti e mantenere i diagrammi creati.
Le tre macrocomponenti verranno descritte in dettaglio in seguito su questo documento.

3.3 Interfaccia REST-like

Per l'interfaccia della componente *back-end*$_G$ si è scelto di utilizzare uno stile basato REST. All'interno di un'unica sessione utente, a partire dall'operazione di login fino a quella di logout, l'interfaccia con cui si accede agli elementi delle collection può considerarsi effettivamente REST.
I motivi che hanno spinto alla scelta di RESTsono:

- Semplicità di utilizzo;
- Facile integrazione con i *framework*$_G$ esistenti;
- Indipendenza dal linguaggio di programmazione utilizzato.

REST utilizza il concetto di risorsa, ovvero un aggregato di dati con un nome (URI) e una rappresentazione, su cui è possibile invocare le operazioni CRUD tramite la seguente corrispondenza:

3.4 Architettura del Server

L'implementazione scelta per il backend dell'applicazione è un server che segue lo stile architetturale *REST*$_G$; ciò implica che:

- l'applicazione renda disponibili le sue funzioni in veste di risorse web;
- ogni risorsa resa disponibile è indirizzabile univocamente utilizzando un indirizzo URL;

- l'interfaccia delle risorse deve essere uniforme e deve garantire un insieme ben definito di operazioni e una gestione priva di stato delle operazioni.

Tale architettura permette l'indipendenza completa tra *back-end*$_G$ e *front-end*$_G$, permettendo così espansioni su altre piattaforme senza dover modificare il *back-end*$_G$ dell'applicazione. Il collegamento tra il *front-end*$_G$ e i modelli nel *back-end*$_G$ verrà implementato da uno stack di *middleware*$_G$.

3.5 Architettura del Client

Il *front-end*$_G$ di SWEDesigner è una *Single Page Application*$_G$ scritta in HTML5, CSS3 e *JavaScript*$_G$.

4 Componenti del *Back-end*

4.1 Descrizione packages e classi

4.1.1 SWEDesigner::Server

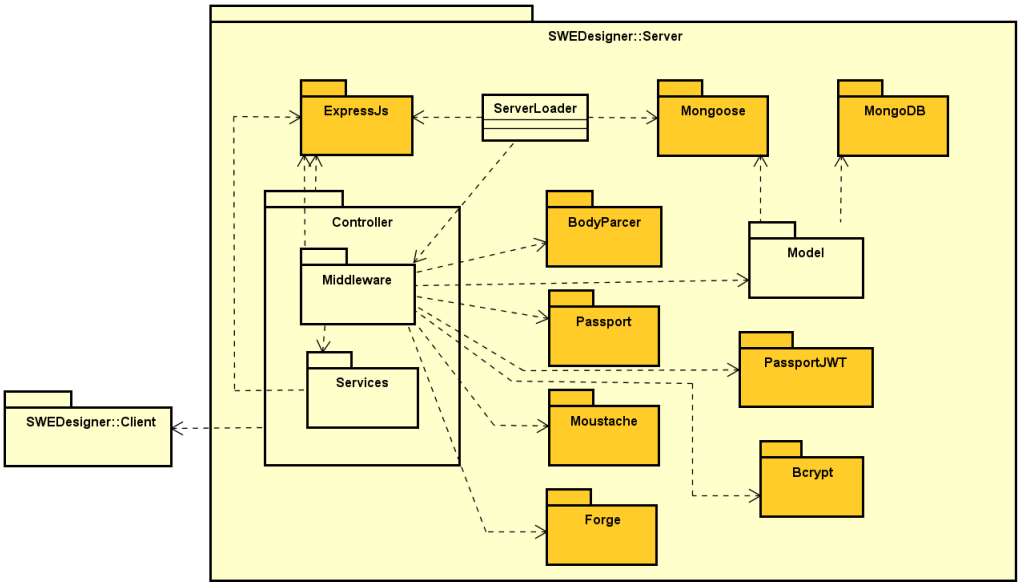

Figura 2: Diagramma dei packages SWEDesigner::Server

4.1.1.1 Informazioni sul Package

- **Descrizione:**
  Package che racchiude tutta la componente del server scritta in JavaScript.
- **Padre:**
  SWEDesigner
- **Package contenuti:**
  - SWEDesigner::Server::Controller;
    Questo package contiene tutte le componenti middleware e i servizi con i quali si interfacciano. Ogni controller si occupa di gestire tutte le richieste del client attraverso i sui componenti middleware e di rispondere ad esse attraverso l'interfaccia REST definita da express.
  - SWEDesigner::Server::Model;
    Questo package contene le classi e i metodi che si interfacciano con il database

passando dal modulo di moongose. Il model si occupa quindi delle richieste al database e delle operazioni ad esso dedicate rispondendo alle varie richieste del controller.

4.1.1.2 Informazioni sulle Classi

- SWEDesigner::Server::ServerLoader
  - **Descrizione:**
    Classe che consente il caricamento del server.
  - **Utilizzo:**
    La classe viene utilizzata per caricare tutte le componenti del server nel momento dell'avvio dell'applicazione.

4.1.2 SWEDesigner::Server::Controller

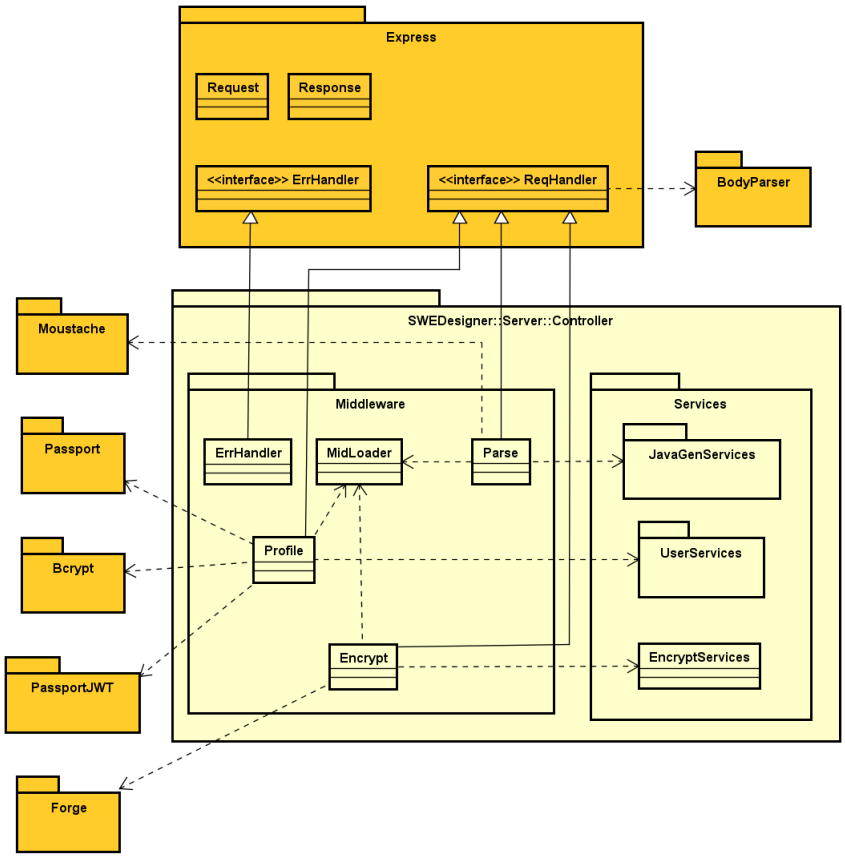

Figura 3: Diagramma dei packages SWEDesigner::Server::Controller

4.1.2.1 Informazioni sul Package

- **Descrizione:**
  Il package racchiude al suo interno tutti i servizi e le componenti middleware che regolano la bussiness logic del server.
- **Padre:**
  SWEDesigner::Server
- **Package contenuti:**
  - *SWEDesigner::Server::Controller::Middleware*;
    Si tratta del package contenente tutte le componenti middleware che rispondono alle varie richieste del client.
  - *SWEDesigner::Server::Controller::Services*;
    Si tratta del package contentente tutti i servizi utilizzati dalle componenti middleware per il corretto svolgersi delle loro operazioni.

4.1.3 SWEDesigner::Server::Controller::Middleware

4.1.3.1 Informazioni sul Package

- **Descrizione:**
  Si tratta del package contenente tutte le componenti middleware che rispondono alle varie richieste del client.
- **Padre:**
  SWEDesigner::Server::Controller

4.1.3.2 Informazioni sulle Classi

- SWEDesigner::Server::Controller::Middleware::ErrHandler
  - **Descrizione:**
    Si tratta della classe che si occupa della gestione degli errori nelle richieste REST che derivano dal client.
  - **Utilizzo:**
    La classe, interfacciandosi con un'interfaccia di express, si occupa di gestire, grazie ad una relazione con la realita interfaccia di express, tutti gli errori delle richieste arrivate dal client.
  - **Relazioni con le altre classi:**
    - *IN* express::ErrHandler
- SWEDesigner::Server::Controller::Middleware::MidLoader

  - **Descrizione:**
    Si tratta della classe che si occupa di caricare tutte le componenti middleware del server.
  - **Utilizzo:**
    La classe carica tutte le componenti middelware del servr applicando il design patter *Facade*$_G$.

- SWEDesigner::Server::Controller::Middleware::Parse
  - **Descrizione:**
    Classe per la gestione del parsing dei file JSON.
  - **Utilizzo:**
    La classe, utilizzando la componente esterna Moustache, si occupa di fare il parsing dei file JSON in arrivo dal client e creare, tramite il servizio JavaGen, il codice sorgente.
  - **Relazioni con le altre classi:**
    * *IN* MidLoader
    * *IN* express::ReqHandler

- SWEDesigner::Server::Controller::Middleware::Profile
  - **Descrizione:**
    Classe per la gestione dei servizi middleware riguardanti l'utente, come registrazione e autenticazione.
  - **Utilizzo:**
    Con l'ausilio di moduli esterni di Passport e Bcrypt, la classe si occupa di gestire tutti quei servizi middleware che riguardano il profilo dell'utente.
  - **Relazioni con le altre classi:**
    * *IN* MidLoader
    * *IN* express::ReqHandler

- SWEDesigner::Server::Controller::Middleware::Encrypt
  - **Descrizione:**
    Classe per l'encrypt e il decrypt dei file di progetto.
  - **Utilizzo:**
    La classe, utilizzando il modulo esterno Forge, encrypta i file di progetto generati tramite SWEDesigner e ne effettua la decrittazione al momento del caricamento degli stessi.
  - **Relazioni con le altre classi:**

* *IN* MidLoader
* *IN* express::ReqHandler

4.1.4 SWEDesigner::Server::Controller::Services

4.1.4.1 Informazioni sul Package

- **Descrizione:**
  Qualcosa
- **Padre:**
  SWEDesigner::Server::Controller
- **Package contenuti:**
  - *SWEDesigner::Server::Controller::Services::JavaGenServices*;
    Tramite questo package si effettuano tutti i servizi dediti alla generaizone di codice sorgente a partire dagli UML creati dall'utente all'intero di SWEDesigner.
  - *SWEDesigner::Server::Controller::Services::UserServices* Il package presenta tutti quei servizi utili alle componenti middleware per la gestione del profilo dell'utente.

4.1.4.2 Informazioni sulle Classi

- SWEDesigner::Server::Controller::Services::EncryptServices
  - **Descrizione:**
    La classe offre il servizio di encrypt e decrypt utile alla componente Middleware::Encrypt.
  - **Utilizzo:**
    Il servizio in questione offre tutti i metodoti utili al middleware per effettuare la criptazione e decrittazione dei file di progetto caricati e salvati dagli utenti.

4.1.5 SWEDesigner::Server::Controller::Services::JavaGenService

4.1.5.1 Informazioni sul Package

- **Descrizione:**
  Il pacchetto contiente tutti i servizi di generazione e download del codice sorgente a parite dagli UML disegnati dall'utente.
- **Padre:**
  SWEDesigner::Server::Controller::Services

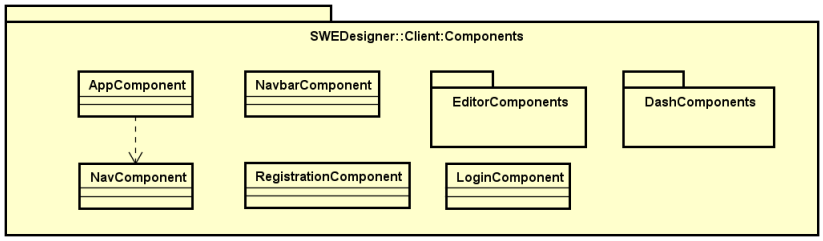

Figura 4: Diagramma dei packages SWEDesigner::Server::Controller::Services::JavaGenService

4.1.5.2 Informazioni sulle Classi

- SWEDesigner::Server::Controller::Services::JavaGenService::JavaGenFactory
  - **Descrizione:**
    La classe implementa il pattern $Factory_G$ per la creazione di un controller che si occupi di gestire i servizi di generazione codice.
  - **Utilizzo:**
    La classe viene utilizzata come creator per la creazione del controller dei servizi di generazione, tramite parsing, e download del codice sorgente.
- SWEDesigner::Server::Controller::Services::JavaGenService::ParseService
  - **Descrizione:**
    La classe effettua il parsing del JSON generato dai diagrammi UML e lo trasforma in codice sorgente Java.
  - **Utilizzo:**
    Mediante l'utilizzo del modulo esterno Moustache, il servizio si occupa di effettuare il parsing del JSON inviato tramite richista REST dal client e di generare il codice sorgente associato.
  - **Relazioni con le altre classi:**
    - *OUT* Express::ReqHandler
- SWEDesigner::Server::Controller::Services::JavaGenService::DownloadService
  - **Descrizione:**
    La classe effettua il download del codice sorgente generato a partire dagli UML dell'utente.
  - **Utilizzo:**
    La classe invia al client che lo ha richiesto lo stream del file .jar sorgente creato a partire dal JSON inviato tramite richiesta REST.
  - **Relazioni con le altre classi:**

* *OUT* Express::ReqHandler

4.1.6 SWEDesigner::Server::Controller::Services::UserServices

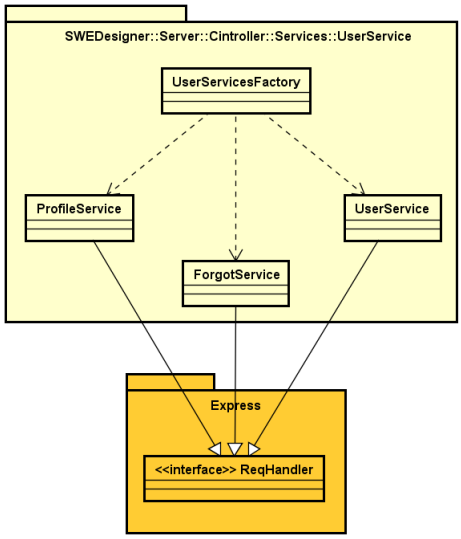

Figura 5: Diagramma dei packages SWEDesigner::Server::Controller::Services::UserServices

4.1.6.1 Informazioni sul Package

- **Descrizione:**
  Il packege contiene al suo interno tutti i servizi utili all'utente per l'autenticazione e la gestione del profilo.
- **Padre:**
  SWEDesigner::Server::Controller::Services

4.1.6.2 Informazioni sulle Classi

- SWEDesigner::Server::Controller::Services::UserServices::UserServicesFactory
  - **Descrizione:**
    La classe implementa il patter $Factory_G$ per la creazione di un controller che gestisca i servizi relativi all'autenticazione e registrazione di un utente.
  - **Utilizzo:**
    La classe viene utilizzata come creator per la creazione del controller che si occupa dei servizi riguardanti il profilo dell'utente e le sue credenziali.
- SWEDesigner::Server::Controller::Services::UserServices::ProfileService

  - **Descrizione:**
    La classe gestisce i servizi di autenticazione e registrazione.
  - **Utilizzo:**
    La classe contiene tutti metodi necessari per l'autenticazione e la registrazione di un utente all'interno del database.
  - **Relazioni con le altre classi:**
    * *OUT* Express::ReqHandler

- SWEDesigner::Server::Controller::Services::UserServices::ForgotService
  - **Descrizione:**
    La classe offre il servizio di recupero password.
  - **Utilizzo:**
    La classe permette all'utente di recuperare le credenziali del proprio account.
  - **Relazioni con le altre classi:**
    * *OUT* Express::ReqHandler

- SWEDesigner::Server::Controller::Services::UserServices::UserService
  - **Descrizione:**
    La classe gestisce il profilo dell'utente.
  - **Utilizzo:**
    La classe offre i servizi di management di un account.
  - **Relazioni con le altre classi:**
    * *OUT* Express::ReqHandler

4.1.7 SWEDesigner::Server::Model

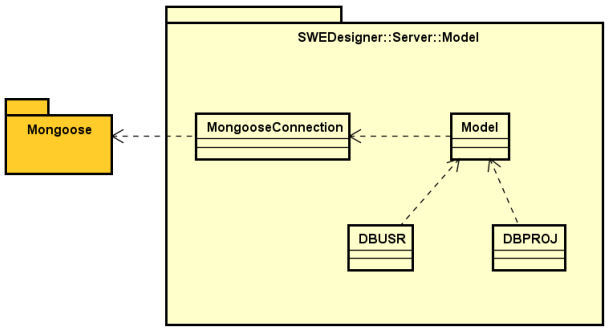

Figura 6: Diagramma dei packages SWEDesigner::Server::Model

4.1.7.1 Informazioni sul Package

- **Descrizione:**
  Il package si occupa delle comunicazioni con il database passando peril modulo di Mongoose per le richieste allo stesso.
- **Padre:**
  SWEDesigner

4.1.7.2 Informazioni sulle Classi

- SWEDesigner::Server::Model::MongooseConnection
  - **Descrizione:**
    La classe stablisce la connessione al database.
  - **Utilizzo:**
    La classe si connette al database passando per il modulo di Mongoose.
- SWEDesigner::Server::Model::Model
  - **Descrizione:**
    La classe implementa il pattern $Facade_G$ per la realizzazione del data tier.
  - **Utilizzo:**
    La classe fornisce un'interfaccia per le classi che gestiscono il database.
- SWEDesigner::Server::Model::DBUSR
  - **Descrizione:**
    La classe gestisce il database utenti.
  - **Utilizzo:**
    La classe fornisce tutti gli strumenti per la lettura e la scrittura di un utente.
- SWEDesigner::Server::Model::DBPROJ
  - **Descrizione:**
    La classe gestisce il database dei progetti.
  - **Utilizzo:**
    La classe fornisce tutti gli strumenti per la lettura e la scrittura di un progetto.

4.2 Scenari

4.2.1 Gestione generale delle richieste

4.2.2 Fallimento vincolo "utente autenticato"

4.2.3 Fallimento vincolo "utente non autenticato"

4.2.4 Richiesta POST /login

4.2.5 Richiesta DELETE /logout

4.3 Descrizione librerie aggiuntive

5 Front-end

5.1 Descrizione packages e classi

5.1.1 SWEDesigner::Client

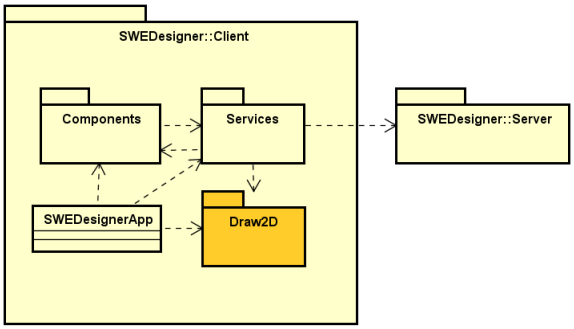

Figura 7: Diagramma dei packages SWEDesigner::Client

5.1.1.1 Informazioni sul Package

- **Descrizione:**
  Package che racchiude tutta la componente di Front-end scritta in JavaScript.
- **Padre:**
  SWEDesigner
- **Package contenuti:**
  - SWEDesigner::Client::Components;
  - SWEDesigner::Client::Services;
  - SWEDesigner::Server;

5.1.1.2 Informazioni sulle Classi

- SWEDesigner::Client::SWEDesignerApp
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

5.1.2 SWEDesigner::Client::Components

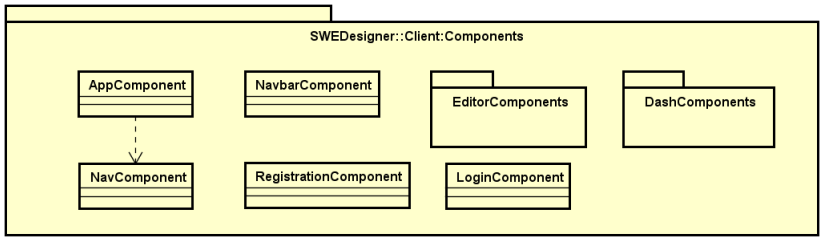

Figura 8: Diagramma dei packages SWEDesigner::Client::Components

5.1.2.1 Informazioni sul Package

- **Descrizione:**
  Qualcosa
- **Padre:**
  SWEDesigner::Client
- **Package contenuti:**
  - SWEDesigner::Client::Components::EditorComponents;
  - SWEDesigner::Client::Components::DashComponents;

5.1.2.2 Informazioni sulle Classi

- SWEDesigner::Client::Components::AppComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa
- SWEDesigner::Client::Components::NavBarComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa
- SWEDesigner::Client::Components::NavComponent
  - **Descrizione:**
    Qualcosa

  - **Utilizzo:**
    Qualcosa

- SWEDesigner::Client::Components::RegistrationComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa
- SWEDesigner::Client::Components::LoginComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

5.1.3 SWEDesigner::Client::Components::EditorComponents

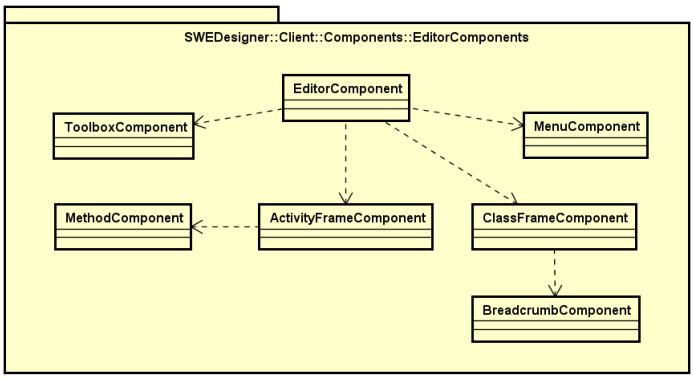

Figura 9: Diagramma dei packages SWEDesigner::Client::Components::EditorComponents

5.1.3.1 Informazioni sul Package

- **Descrizione:**
  Qualcosa
- **Padre:**
  SWEDesigner::Client::Components

5.1.3.2 Informazioni sulle Classi

- SWEDesigner::Client::Components::EditorComponents::ToolboxComponent

  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

- SWEDesigner::Client::Components::EditorComponents::EditorComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

- SWEDesigner::Client::Components::EditorComponents::MenuComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

- SWEDesigner::Client::Components::EditorComponents::MethodComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

- SWEDesigner::Client::Components::EditorComponents::ActivityFrameComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

- SWEDesigner::Client::Components::EditorComponents::ClassFrameComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

- SWEDesigner::Client::Components::EditorComponents::BreadcrumbComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

5.1.4 SWEDesigner::Client::Components::DashComponents

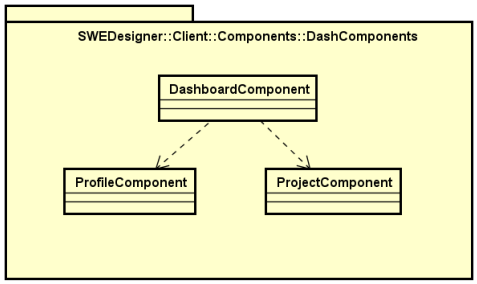

Figura 10: Diagramma dei packages SWEDesigner::Client::Components::DashComponents

5.1.4.1 Informazioni sul Package

- **Descrizione:**
  Qualcosa
- **Padre:**
  SWEDesigner::Client::Components

5.1.4.2 Informazioni sulle Classi

- SWEDesigner::Client::Components::DashComponents::DashComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa
- SWEDesigner::Client::Components::DashComponents::ProfileComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa
- SWEDesigner::Client::Components::DashComponents::ProjectComponent
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

5.1.5 SWEDesigner::Client::Service

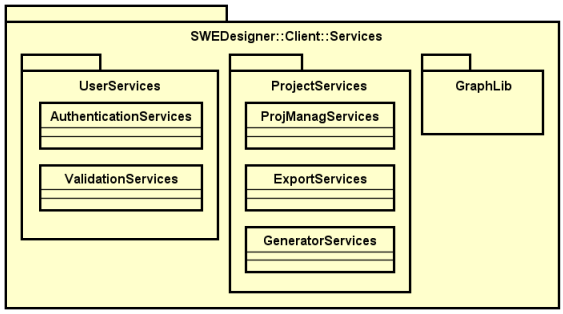

Figura 11: Diagramma dei packages SWEDesigner::Client::Service

5.1.5.1 Informazioni sul Package

- **Descrizione:**
  Qualcosa
- **Padre:**
  SWEDesigner::Client
- **Package contenuti:**
  - SWEDesigner::Client::Services::UserServices;
  - SWEDesigner::Client::Services::ProjectServices;
  - SWEDesigner::Client::Services::GraphLib;

5.1.6 SWEDesigner::Client::Services::UserServices

5.1.6.1 Informazioni sul Package

- **Descrizione:**
  Qualcosa
- **Padre:**
  SWEDesigner::Client::Services

5.1.6.2 Informazioni sulle Classi

- SWEDesigner::Client::Services::UserServices::AuthenticationService
  - **Descrizione:**
    Qualcosa

  - **Utilizzo:**
    Qualcosa

- SWEDesigner::Client::Services::UserServices::ValidationServices
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

5.1.7 SWEDesigner::Client::Services::ProjectServices

5.1.7.1 Informazioni sul Package

- **Descrizione:**
  Qualcosa
- **Padre:**
  SWEDesigner::Client::Services

5.1.7.2 Informazioni sulle Classi

- SWEDesigner::Client::Services::ProjectServices::ProjManagServices
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa
- SWEDesigner::Client::Services::ProjectServices::ExportServices
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa
- SWEDesigner::Client::Services::ProjectServices::GeneratorServices
  - **Descrizione:**
    Qualcosa
  - **Utilizzo:**
    Qualcosa

6 Diagrammi delle attività

6.1 Applicazione SWEDwsigner

6.1.1 Attività principali

6.1.2 Registrazione

6.1.3 Recupero password

6.1.4 Login

6.1.5 Modifica profilo

6.1.6 Altro

7 Stime di fattibilità e di bisogno e di risorse

8 Design pattern

8.1 Design Pattern Architetturali

8.1.1 MVVM

8.1.2 Dependency Injection

8.2 Design Pattern Creazionali

8.2.1 Factory ad esempio

8.3 Design Pattern Strutturali

8.3.1 Decorator

8.3.2 Facede

8.4 Design Pattern Comportamentali

8.4.1 Observer

8.4.2 Command

9 Tracciamento

9.1 Tracciamento componenti - requisiti

9.2 Tracciamento requisiti - componenti

10 Appendici

A Descrizione Design Pattern

I *Design Pattern*$_G$ sono un modello logico da applicare per la soluzione di problemi ricorrenti. L'impiego di questi modelli rende l'architettura più manutenibile. Verranno di seguito illustrati i Design Pattern implementati nella costruzione dell'architettura di alto livello, divisi per categoria di applicazione:

A.1 Design Pattern Architetturali

A.1.1 MVVM

- **Scopo**: Lo scopo del MVVM è quello di separare le seguenti componenti in modo da non mescolare il codice della logica con quella dell'interfaccia utente:
  - Model - rappresenta il punto di accesso ai dati. Trattasi di una o più classi che leggono dati dal DB, oppure da un servizio Web di qualsivoglia natura;
  - View - rappresenta la vista dell'applicazione, l'interfaccia grafica che mostrerà i dati;
  - ViewModel - è il punto di incontro tra la View e il Model: i dati ricevuti da quest'ultimo sono elaborati per essere presentati e passati alla View.
- **Motivazione**: MVVM è stato progettato per utilizzare le funzioni di binding dei dati in WPF (Windows Presentation Foundation) per facilitare la separazione della view dal resto del modello, rimuovendo praticamente tutti i codici GUI_G dal livello di visualizzazione. Invece di richiedere agli sviluppatori di *user experience*$_G$ (UX) di scrivere il codice GUI_G, possono utilizzare il linguaggio di marcatura dei framework e creare connessioni di dati al modello di visualizzazione, che viene scritto e gestito dagli sviluppatori di applicazioni. La separazione dei ruoli consente ai progettisti interattivi di concentrarsi sulle esigenze UX anziché sulla programmazione della logica aziendale. Gli strati di un'applicazione possono quindi essere sviluppati in più flussi di lavoro per una maggiore produttività. Anche quando un singolo sviluppatore lavora sull'intera base di codice, una corretta separazione della vista dal modello è più produttiva, poiché l'interfaccia utente tipicamente cambia ed è in ritardo nel ciclo di sviluppo in base ai feedback degli utenti finali.
- **Applicabilità**: il modello MVVM è in definitiva la moderna struttura del modello MVC, quindi l'obiettivo principale è sempre lo stesso per fornire una netta separazione tra logica di dominio e livello di presentazione. Esso è applicabile nei seguenti casi:

– Quando si vuole ottenere la vera separazione tra la view e la model oltre a conseguire la separazione e l'efficienza che si guadagna ad averla;

– Quando si vuole codice manutenibile e estensibile.

A.1.2 Three-Tier

- **Scopo**: tale design pattern permette una disgiunzione tra i vari gruppi di entità che cooperano nell'erogazione del servizio. Esisterà un livello che si occuperà di interagire con il cliente offrendo l'interfaccia grafica, un altro livello che gestirà di eseguire la parte algoritmica dell'applicazione e un altro livello che si occuperà di persistere i dati e recuperarli. Ogni livello comunicherà solo con i livelli adiacenti.
- **Motivazione**: si tratta di un pattern molto utilizzato nelle applicazioni web perché rende l'applicazione flessibile, riutilizzabile e scalabile. Con la separazione di un'applicazione in livelli, gli sviluppatori, per modificare o aggiungere funzionalità, possono infatti modificare solo uno specifico livello piuttosto che dover riscrivere l'intera applicazione. Ciò garantisce dunque una maggiore semplicità di progettazione/implementazione secondo la filosofia del divide et impera ed una maggiore manutenibilità.

A.2 Design Pattern Creazionali

A.2.1 Factory Method

- **Scopo**: il design pattern Factory Method indirizza il problema della creazione di oggetti senza specificarne l'esatta classe, fornendo un'interfaccia per creare un oggetto, ma lasciando che le sottoclassi decidano quale oggetto istanziare. Definisce un'interfaccia (Creator) per ottenere una nuova istanza di un oggetto (Product). Delega ad una classe derivata (ConcreteCreator) la scelta di quale classe istanziare (ConcreteProduct).
- **Motivazione**: la creazione di un oggetto può, spesso, richiedere processi complessi la cui collocazione all'interno della classe di composizione potrebbe non essere appropriata. Esso può, inoltre, comportare duplicazione di codice, richiedere informazioni non accessibili alla classe di composizione, o non fornire un sufficiente livello di astrazione. Il Factory Method indirizza questi problemi definendo un metodo separato per la creazione degli oggetti. Tale metodo può essere ridefinito dalle sottoclassi per definire il tipo derivato di prodotto che verrà effettivamente creato.
- **Applicabilità**: tale design pattern verrà utilizzato nei seguenti casi:

- si desidera che la creazione di un oggetto non precluda il suo riuso senza una significativa duplicazione di codice;
- si desidera che la creazione di un oggetto non richieda l'accesso ad informazioni o risorse che non dovrebbero essere contenute nella classe di composizione;
- si desidera che la gestione del ciclo di vita degli oggetti gestiti debba essere centralizzata in modo da assicurare un comportamento consistente all'interno dell'applicazione.

A.3 Design Pattern Strutturali

A.3.1 Decorator

- **Scopo**: aggiungere dinamicamente responsabilità a un oggetto. I Decorator forniscono un'alternativa flessibile alla definizione di sottoclassi come strumento per l'estensione delle funzionalità;
- **Motivazione**: talvolta si vogliono aggiungere responsabilità a singoli oggetti e non a un'intera classe. Un modo per aggiungere responsabilità consiste nel racchiudere il componente da decorare in un altro. L'oggetto contenitore è chiamato Decorator. Il Decorator ha un interfaccia conforme a quella dell'elemento decorato, in modo da rendere trasparente la sua presenza ai client. Il decorator trasferisce le richieste al componente decorato e può svolgere azioni aggiuntive prima o dopo il trasferimento della richiesta;
- **Applicabilità**: il pattern Decorator può essere utilizzato nei seguenti casi:
  - Si vuole poter aggiungere responsabilità a singoli oggetti dinamicamente ed in modo trasparente;
  - Si vuole poter togliere responsabilità agli oggetti;
  - Si vuole definire un gran numero di estensioni indipendenti.

A.3.2 Facade

- **Scopo**: fornire un'interfaccia unificata per un insieme di interfacce presenti in un sottosistema. Definisce un'interfaccia di livello più alto che rende il sottosistema più semplice da utilizzare;
- **Motivazione**: suddividere un sistema in sottosistemi aiuta a ridurne la complessità. Un obiettivo comune di progettazione è la minimizzazione delle comunicazioni e delle dipendenza fra i diversi sottosistemi. Un modo per raggiungere questo obiettivo è introdurre un oggetto facade, che fornisce un'interfaccia unica e semplificata per accedere alle funzionalità offerte da un sottosistema;

- **Applicabilità**: il pattern Facade può essere utilizzato nei seguenti casi:
  - Quando si vuole fornire un'interfaccia semplice a un sottosistema complesso poiché fornisce una vista semplice di base su un sottosistema che si rivela essere sufficiente per la maggior parte dei client;
  - Nei casi in cui ci sono molte dipendenze fra i client e le classi che implementano un'astrazione in quanto si disaccoppia il sottosistema dai client e dagli altri sistemi, promuovendo portabilità e indipendenza dei sottosistemi;
  - Quando si vogliono organizzare i sottosistemi in una struttura a livelli.

A.4 Design Pattern Comportamentali

A.4.1 Dependency Injection

- **Scopo**: Il Dependency Injection è un Design Pattern che permette la separazione del comportamento degli oggetti dalla loro dipendenze. Invece di istanziare le classi in modo diretto ogni componente riceve i riferimenti agli altri componenti necessari come parametri nel costruttore. Un utilizzo comune è quello con i plugin che vengono caricati dinamicamente. Gli elementi coinvolti sono:
  - Un dipendente consumatore;
  - Una dichiarazione delle dipendenze tra la componenti, definita come contratto di un interfaccia;
  - Un injector che crea istanze di classi che implementano una data dipendenza su richiesta.

  Il dependent object dichiara da quali componenti dipende. L'injector decide quali classi soddisfano suoi requisiti e in caso affermativo gliele fornisce. Questa operazione può avvenire anche a runtime. Questo è un chiaro vantaggio poiché possono essere create dinamicamente diverse implementazioni di un componente software da passare allo stesso test. In questo modo il test può testare componenti diverse senza sapere che le loro implementazioni sono diverse.
- **Motivazione**: lo scopo principale di questo pattern è quello di permettere una selezione a runtime su più implementazioni di una interfaccia dipendente. È particolarmente utile per fornire delle implementazioni di stub per componenti complesse, ma anche per gestire i plugin e per inizializzare servizi software. I test di unità comportano delle problematiche, poiché spesso richiedono la presenza di una parte di infrastruttura non ancora implementata. Il Dependency Injection semplifica il processo di testing per un istanza isolata. Poiché le componenti dichiarano le proprie dipendenze, un test può automaticamente istanziare le componenti necessarie.

- **Applicabilità**: Di seguito vengono elencati tre modi con cui un oggetto può ricevere un riferimento da un modulo esterno:
  - Interface injection: l'oggetto fornisce un interfaccia che gli utenti possono implementare in modo da ottenere a runtime le dipendenze;
  - Setter injection: il dependent module espone un metodo setter che il frameworkG usa per iniettarvi le dipendenze;
  - Constructor injection: le dipendenze vengono fornite tramite il costruttore della classe.

A.4.2 Command

- **Scopo**: Incapsula una richiesta in un oggetto, consentendo di parametrizzare i client con richieste diverse, accodare o mantenere uno storico delle richieste e gestire richieste cancellabili;
- **Motivazione**: Talvolta è necessario inoltrare richieste a oggetti senza conoscere nulla dell'operazione richiesta o del destinatario della richiesta. Il pattern Command permette agli oggetti dell'ambiente di inoltrare richieste a oggetti sconosciuti dell'applicazione trasformando la richiesta in un oggetto;
- **Applicabilità**: Il pattern Command può essere utilizzato nei seguenti casi:
  - Per parametrizzare gli oggetti rispetto a un'azione da compiere;
  - Per specificare, accordare ed eseguire le richieste in tempi diversi;
  - Per consentire l'annullamento di operazioni;
  - Per organizzare un sistema in operazioni d'alto livello a loro volta basate su operazioni primitive.