Specifica Tecnica

Gruppo SWEet BIT — Progetto SWEDesigner

Informazioni sul documento

Versione	2.0.0
Redazione	Bertolin Sebastiano
	Massignan Fabio
	Santimaria Davide
Verifica	Salmistraro Gianmarco
Approvazione	Salvatore Pilò
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
2.0.0	2017/06/22	Salvatore Pilò	Approvazioe documento
1.1.3	2017/06/18	Salmistraro Gianmarco	Verifica documento
1.0.3	2017/06/15	Bertolin Sebastiano	Correzione e ampliamento sezione Front-End
1.0.2	2017/06/08	Santimaria Davide	Correzione e ampliamento sezione Back-End
1.0.1	2017/06/07	Massignan Fabio	Revisione e correzione errori minori
1.0.0	2017/05/07	Bertolin Sebastiano	Approvazione Documento
0.1.0	2017/05/07	Massignan Fabio	Verifica Documento
0.0.4	2017/05/02	Bodian Malick	Stesura sezione Descrizione architettura
0.0.3	2017/04/30	Bertolin Sebastiano	Stesura sezione Tecnologie utilizzate
0.0.2	2017/04/29	Pilò Salvatore	Stesura sezione Introduzione
0.0.1	2017/04/27	Salmistraro Gianmarco	Creazione struttura documento

Indice

1 Introduzione **6**
1.1 Scopo del documento . . . 6
1.2 Scopo del prodotto . . . 6
1.3 Glossario . . . 6
1.4 Riferimenti . . . 6
1.4.1 Normativi . . . 6
1.4.2 Informativi . . . 7

2 Tecnologie utilizzate **8**
2.1 Server . . . 8
2.1.1 Node.js . . . 8
2.1.1.1 Vantaggi . . . 8
2.1.1.2 Svantaggi . . . 8
2.1.2 Expressjs . . . 9
2.1.2.1 Vantaggi . . . 9
2.1.2.2 Svantaggi . . . 9
2.1.3 MongoDB . . . 9
2.1.3.1 Vantaggi . . . 9
2.1.3.2 Svantaggi . . . 10
2.1.4 Mongoose . . . 10
2.1.4.1 Vantaggi . . . 10
2.1.4.2 Svantaggi . . . 10
2.1.5 Grunt . . . 10
2.1.5.1 Vantaggi . . . 11
2.1.5.2 Svantaggi . . . 11
2.2 Librerie . . . 11
2.2.1 Mustache . . . 11
2.2.1.1 Vantaggi . . . 12
2.2.1.2 Svantaggi . . . 12
2.2.2 Passport: . . . 12
2.2.2.1 Vantaggi . . . 12
2.2.2.2 Svantaggi . . . 12
2.2.3 BodyParser: . . . 12
2.2.3.1 Vantaggi . . . 13
2.2.3.2 Svantaggi . . . 13
2.2.4 Forge: . . . 13
2.2.4.1 Vantaggi . . . 13
2.2.4.2 Svantaggi . . . 13
2.2.5 PassportJWT: . . . 13
2.2.5.1 Vantaggi . . . 13
2.2.5.2 Svantaggi . . . 13

2.2.6 Bcrypt: . . . 14
2.2.6.1 Vantaggi . . . 14
2.2.6.2 Svantaggi . . . 14
2.3 Client . . . 14
2.3.1 Angular 4.0 . . . 14
2.3.1.1 Vantaggi . . . 14
2.3.1.2 Svantaggi . . . 14
2.3.2 JointJS . . . 15
2.3.2.1 Vantaggi . . . 15
2.3.2.2 Svantaggi . . . 15
2.3.3 HTML5 . . . 15
2.3.3.1 Vantaggi . . . 15
2.3.3.2 Svantaggi . . . 15
2.3.4 CSS3 . . . 16
2.3.4.1 Vantaggi . . . 16

3 Descrizione architettura **17**
3.1 Metodo e formalismo di specifica . . . 17
3.2 Architettura generale . . . 17
3.3 Interfaccia REST-like . . . 18
3.4 Architettura del Server . . . 18
3.5 Architettura del Client . . . 19

4 Componenti del *Back-end* **20**
4.1 Descrizione packages e classi . . . 20
4.1.1 SWEDesigner::Server . . . 20
4.1.1.1 Informazioni sul Package . . . 20
4.1.1.2 Informazioni sulle Classi . . . 21
4.1.2 SWEDesigner::Server::Controller . . . 21
4.1.2.1 Informazioni sul Package . . . 21
4.1.3 SWEDesigner::Server::Controller::Middleware . . . 21
4.1.3.1 Informazioni sul Package . . . 21
4.1.3.2 Informazioni sulle Classi . . . 22
4.1.4 SWEDesigner::Server::Controller::Services . . . 24
4.1.4.1 Informazioni sul Package . . . 24
4.1.4.2 Informazioni sulle Classi . . . 24
4.1.5 SWEDesigner::Server::Controller::Services::UserServices . . . 25
4.1.5.1 Informazioni sul Package . . . 25
4.1.5.2 Informazioni sulle Classi . . . 25
4.1.6 SWEDesigner::Server::Model . . . 26
4.1.6.1 Informazioni sul Package . . . 27
4.1.6.2 Informazioni sulle Classi . . . 27

5 Front-end **29**

5.1 Descrizione packages e classi 29
5.1.1 SWEDesigner::Client 29
5.1.1.1 Informazioni sul Package 29
5.1.2 SWEDesigner::Client::Components 30
5.1.2.1 Informazioni sul Package 30
5.1.2.2 Informazioni sulle Classi 31
5.1.3 SWEDesigner::Client::Components::Editor 32
5.1.3.1 Informazioni sul Package 32
5.1.3.2 Informazioni sulle Classi 33
5.1.4 SWEDesigner::Client::Components::Menu 34
5.1.4.1 Informazioni sul Package 34
5.1.4.2 Informazioni sulle Classi 35
5.1.5 SWEDesigner::Client::Components::ActivityFrame 37
5.1.5.1 Informazioni sul Package 37
5.1.5.2 Informazioni sulle Classi 37
5.1.6 SWEDesigner::Client::Service 38
5.1.6.1 Informazioni sul Package 38
5.1.6.2 Informazioni sulle Classi 38
5.1.7 SWEDesigner::Client::Service::Models 41
5.1.7.1 Informazioni sul Package 41
5.1.7.2 Informazioni sulle Classi 42

6 Tracciamento **45**
6.1 Tracciamento componenti - requisiti 45
6.2 Tracciamento requisiti - componenti 48

A Descrizione *Design Pattern*$_G$ **53**
A.1 *Design Pattern*$_G$ Architetturali 53
A.1.1 MVVM 53
A.1.2 Three-Tier 54
A.2 *Design Pattern*$_G$ Creazionali 55
A.2.1 Factory Method 55
A.3 *Design Pattern*$_G$ Strutturali 56
A.3.1 Decorator 56
A.3.2 Facade 57
A.4 *Design Pattern*$_G$ Comportamentali 58
A.4.1 Dependency Injection 58
A.4.2 Command 59

Elenco delle figure

1 Diagramma di *deployment*$_G$ per l'architettura 18
2 Diagramma dei packages SWEDesigner::Server 20
3 Diagramma dei packages SWEDesigner::Server::Controller 22
4 Diagramma dei packages SWEDesigner::Server::Controller::Services::UserServices 25
5 Diagramma dei packages SWEDesigner::Server::Model 26
6 Esempio di funzionamento dell'applicazione lato client 29
7 Diagramma dei packages SWEDesigner::Client 29
8 Diagramma dei packages SWEDesigner::Client::Components 30
9 Diagramma dei packages SWEDesigner::Client::Components::Editor . . . 32
10 Diagramma dei packages SWEDesigner::Client::Components::Menu 34
11 Diagramma dei packages SWEDesigner::Client::Service 38
12 Diagramma dei packages SWEDesigner::Client::Service::Models 41
13 Diagramma del *Design Pattern*$_G$ MVVM 53
14 Diagramma del *Design Pattern*$_G$ Three-Tier 54
15 Diagramma del *Design Pattern*$_G$ Factory method 55
16 Diagramma del *Design Pattern*$_G$ Decorator 56
17 Diagramma del *Design Pattern*$_G$ Facade 57
18 Diagramma del *Design Pattern*$_G$ Dependency Injection 58
19 Diagramma del *Design Pattern*$_G$ Command 59

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*3.0.0. 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-1/2016/Progetto/C6p.pdf;
- **Norme di Progetto:** *Norme di Progetto v3.0.0.*
- **Analisi dei Requisiti:** *Analisi dei Requisiti v3.0.0.*

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-v6.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 6.10**: 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 4.0**: *framework*$_G$ *JavaScript*$_G$ per 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 efficiente.

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

- **Programmazione Asincrona**: nonostante questo aspetto è stato inserito anche nei vantaggi, la natura asincrona rende più complessa la comprensione.

2.1.2 Expressjs

Expressjs 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*$_G$ *Expressjs*$_G$ 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.

2.1.2.1 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.

2.1.2.2 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.

2.1.3 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.

2.1.3.1 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;
- Permette di definire query complesse utilizzando un linguaggio che non è *SQL*$_G$;
- Permette di processare parallelamente i dati (*Map-Reduce*$_G$);

2.1.3.2 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;
- **Problemi di concorrenza**: per le operazioni di scrittura viene creato un lock sull'intero database. Questo lock blocca anche le operazioni di lettura.

2.1.4 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.

2.1.4.1 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 intuitiva, ciò dovuto anche dalla sintassi dichiarativa della libreria stessa.

2.1.4.2 Svantaggi

- Non sono stati rilevati particolari svantaggi.

2.1.5 Grunt

Si tratta di un JavaScript Task Runner che automatizza la compilazione e l'esecuzione dei test di unità. Mediante l'utilizzo di un *gruntfile* è possibile pianificare delle attività

che verranno svolte in maniera automatizzata in maniera tale da ottenere un notevole risparmio di tempo sul lavoro.
Tale strumento si è rivelato piuttosto utile per riuscire ad integrare l'utilizzo della libreria grafica JointJS all'interno di Angular 4 in quanto semplifica e automatizza il building del progetto.

2.1.5.1 Vantaggi

- Utilizzando Grunt è possibile eseguire facilmente la compilazione e la verifica dei file;
- Grunt unifica i flussi di lavoro degli sviluppatori web;
- Puoi facilmente lavorare con un nuovo codice base utilizzando Grunt perché contiene meno infrastrutture;
- Accelera il flusso di lavoro di sviluppo e migliora le prestazioni dei progetti.

2.1.5.2 Svantaggi

- Ogni volta che i pacchetti npm vengono aggiornati, è necessario attendere l'aggiornamento dell'autore del Grunt.
- Ogni attività è progettata per fare un lavoro specificato. Se si desidera estendere un'operazione specificata, è necessario utilizzare alcuni trucchi per ottenere il lavoro svolto.
- Grunt include un gran numero di parametri di configurazione per singoli plugin. Di solito, i file di configurazione Grunt sono più lunghi.

2.2 Librerie

Vengono di seguito descritte le *librerie*$_G$ aggiuntive utilizzate dal *back-end*$_G$. La scelta è stata effettuata cercando di valutare la diffusione, il livello di stabilità, l'assenza di errori noti.

2.2.1 Mustache

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

2.2.1.1 Vantaggi

- Mustache fornisce una sintassi pulita e facile da comprendere. Avere una sintassi leggibile è sempre un vantaggio enorme, in quanto ciò significa una manutenzione più facile e una maggiore leggibilità del codice;
- I templates di Mustache possono essere compilati in file JS, così facendo possono essere direttamente carcati;
- E' popolare, quindi nel web è presente una grande comunità in grado di consigliare e aiutare su ogni problema.

2.2.1.2 Svantaggi

- Non sono stati rilevati particolari svantaggi.

2.2.2 Passport:

È un *middleware*$_G$ di autenticazione per *Node.js*$_G$. Estremamente flessibile e modulare, Passport può essere facilmente inserito in qualsiasi applicazione web basata su *Expressjs*$_G$.

2.2.2.1 Vantaggi

- Passport è facile da implementare con Express;
- E' molto flessibile e modulare;
- Permette la serializzazione dell'utente autenticato.

2.2.2.2 Svantaggi

- Non sono stati rilevati particolari svantaggi.

2.2.3 BodyParser:

Modulo$_G$ di terze parti per la corretta lettura delle informazioni contenute nel body delle richieste *HTTP*$_G$; viene utilizzato come *middleware*$_G$ per Expressjs e si occupa della corretta lettura delle informazioni contenute nel body di una richiesta *HTTP*$_G$. Nel nostro caso verrà impiegato per la lettura dei dati del body in formato *JSON*$_G$.

2.2.3.1 Vantaggi

- Permette di memorizzare nel middleware una determinata richiesta del body, e di visualizzarla tramite il comando req.body.

2.2.3.2 Svantaggi

- Non prevede il salvataggio di richieste multiple da parte di più body.

2.2.4 Forge:

È un *modulo*$_G$ in *Javascript*$_G$ che fornisce funzionalità di criptografia per la sicurezza e un set di strumenti per la realizzazione di *Web app*$_G$.

2.2.4.1 Vantaggi

- Permette la creazione di code e sincronizzazione di attività in un'applicazione Web.
- Si basa su AES per la criptazione e la decriptazione in modalità CBC.

2.2.4.2 Svantaggi

- Non sono stati rilevati particolari svantaggi.

2.2.5 PassportJWT:

È un *modulo*$_G$ che fornisce una strategia Passport(§3.2) per l'autenticazione con un *JSON*$_G$ Web Token, che è una tecnica compatta per trasmettere in modo sicuro le informazioni tra 2 oggetti *JSON*$_G$.

2.2.5.1 Vantaggi

- Necessita di poche query, quindi i tempi di risposta sono più rapidi;

2.2.5.2 Svantaggi

- Funziona con solo una chiave, quindi c'è maggiore possibilità per un utente malintenzionato di capire la criptazione.

2.2.6 Bcrypt:

È una *libreria*$_G$ in *Javascript*$_G$ che utilizza la funzione di *hash*$_G$ing per criptare le password.

2.2.6.1 Vantaggi

- E' lento a calcolare l'hash, questo perchè così facendo scoraggia l'attaccante;

2.2.6.2 Svantaggi

- Non sono stati rilevati particolari svantaggi;

2.3 Client

2.3.1 Angular 4.0

Angular 4.0 è un *framework*$_G$ web *open source*$_G$ per lo sviluppo di applicazioni Web lato client; utile a semplificare la realizzazione di applicazioni web, come ad esempio le Single Page Application, cioè applicazioni le cui risorse vengono caricate dinamicamente su richiesta, senza necessità di ricaricare l'intera pagina.

2.3.1.1 Vantaggi

- **Velocità**: Riduce in maniera considerevole il codice necessario a realizzare applicazioni *HTML*$_G$/*JavaScript*$_G$;
- **Ampia documentazione disponibile**;
- **Data Binding bidirezionale**: approccio automatico per aggiornare la vista ogniqualvolta il model cambia e viceversa. Ciò semplifica lo sviluppo eliminando la necessità di manipolare il *DOM*$_G$;
- **Sviluppato per facilitare la fase di test**;
- **Direttive**: caratteristica peculiare di Angular e permettono di estendere la sintassi *HTML*$_G$, creando dei componenti specifici per la propria applicazione e facilmente riutilizzabili;

2.3.1.2 Svantaggi

- **Maggior studio**: curva di apprendimento più ripida rispetto ad altri *framework*$_G$;
- **Codice articolato**: ciò potrebbe comportare, in caso di variazione dei requisiti, delle difficoltà delle successive modifiche.

2.3.2 JointJS

JoniJS è una libreria JavaScript per la creazione di diagrammi e grafici. I diagrammi possono essere completamente interattivi. L'API di JointJS è adatto sia per l'implementazione di uno strumento di diagramma sia per la semplice pubblicazione di diagrammi.

2.3.2.1 Vantaggi

- Permette di collegare oggetti vettoriali con vari tipi di frecce;
- Possibilità di interagire con connessioni e oggetti;
- Elementi pronti per l'uso di schemi ben noti (per lo sviluppo di questo progetto interessa soprattutto UML);
- Diagrammi gerarchici;
- Possibilità di personalizzazione.

2.3.2.2 Svantaggi

- Non sono stati rilevati particolari svantaggi.

2.3.3 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.

2.3.3.1 Vantaggi

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

2.3.3.2 Svantaggi

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

2.3.4 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.

2.3.4.1 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***$_G$.

L'architettura proposta segue il *Design Pattern*$_G$ *Three-tier*. Esso è diviso in 3 livelli:

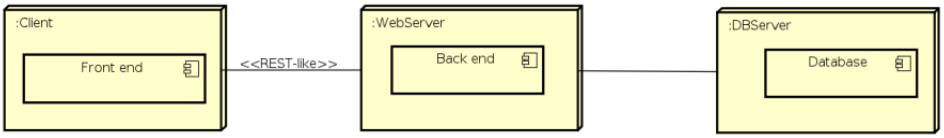

Figura 1: Diagramma di *deployment*$_G$ per l'architettura

- **Presentation tier**: rappresenta l'interfaccia verso l'utente ovvero il *front-end*$_G$;
- **Business logic**: è il livello che coordina l'applicazione ed effettua quindi le decisioni logiche e le valutazioni ovvero il *back-end*$_G$;
- **Data tier**: è il livello dove le informazioni vengono salvate, ovvero il *database*$_G$.

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 REST sono:

- 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*$_G$ realizzata nel framework Angular4.0. L'architettura è MVVM (*Model-View-View Model*), sono descritti due package principali Components e Services, i Components costituiscono la parte Viewmodel dell'architettura, contiene classi atte a rendere dinamica la pagina web; il package Services, che ripecchia la parte Model, offre metodi per l'interazione con il lato server e la libreria grafica. La parte View non viene descritta in quanto composta da template html statici che sono strettamente legati omonimi components.

4 Componenti del *Back-end*

4.1 Descrizione packages e classi

4.1.1 SWEDesigner::Server

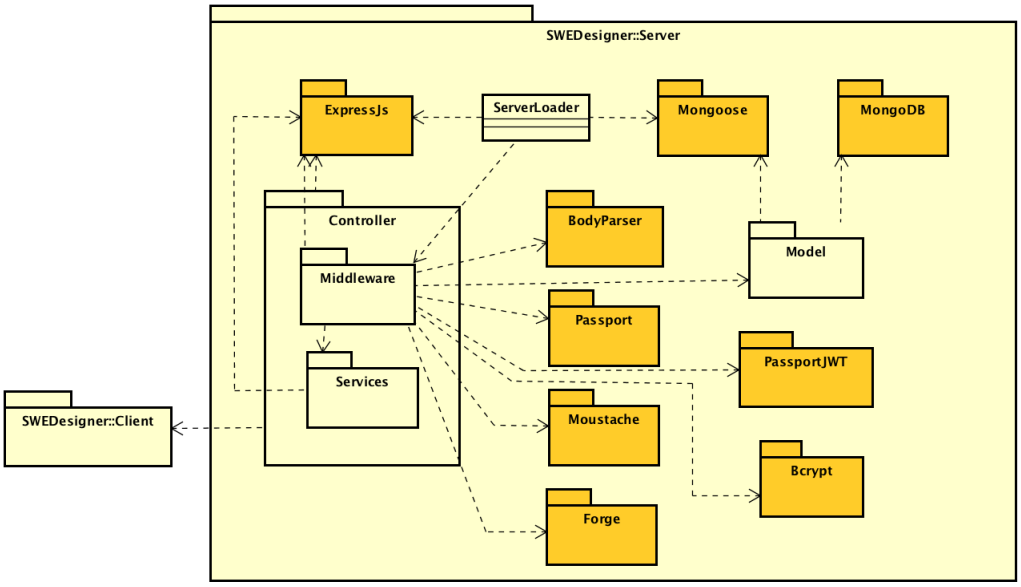

Figura 2: Diagramma dei packages SWEDesigner::Server

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.
    In particolare crea o si connette al database, crea o carica i parametri crittografici dal database e compila (inserisce in cache) il template di Moustache.

4.1.2 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 contenente 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.

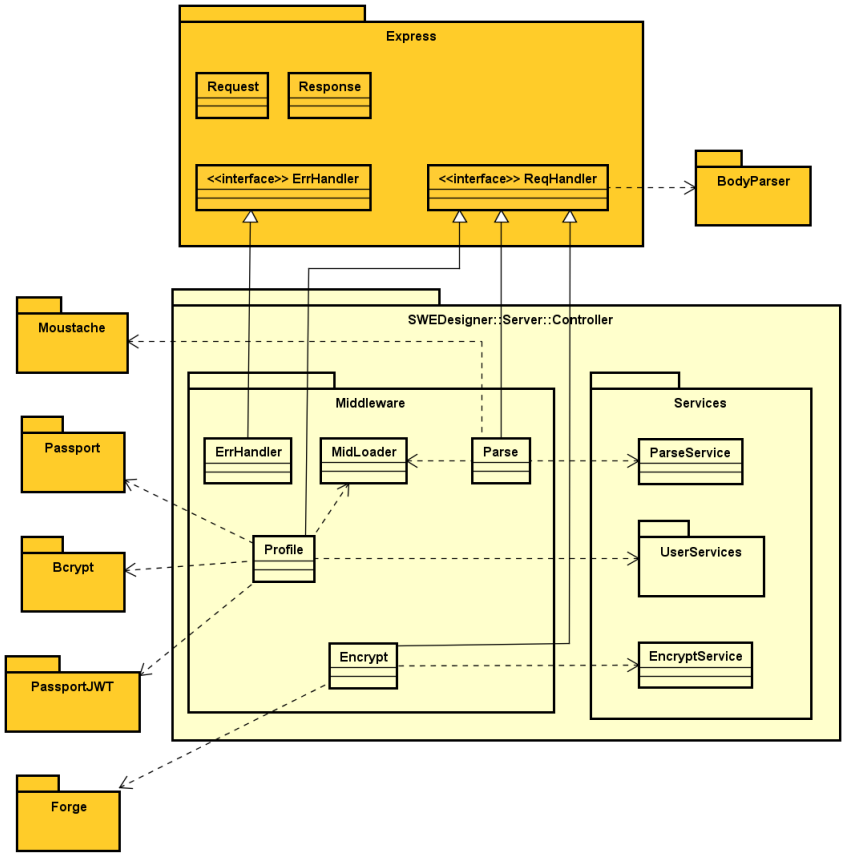

Figura 3: Diagramma dei packages SWEDesigner::Server::Controller

- **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 relativa 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 del middleware utili da pre-caricare.
    In particolare si occupa di istanziare e compilare, quindi caricare in cache, il template di Moustahce per il parsing.
  - **Utilizzo:**
    La classe si occupa di istanziare e caricare i template per il servizio di parsing.
- 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 parseService, 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, con

chiave AES, 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:**
  Package contenente tutti i servizi utili a rispondere allae richiste REST del client.s
- **Padre:**
  SWEDesigner::Server::Controller
- **Package contenuti:**
  - *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::EncryptService
  - **Descrizione:**
    La classe offre il servizio di encrypt e decrypt utile alla componente Middleware::Encrypt.
  - **Utilizzo:**
    Il servizio in questione offre tutti i metodi utili al middleware per effettuare la criptazione e decrittazione dei file di progetto caricati e salvati dagli utenti.
- SWEDesigner::Server::Controller::Services::parseService
  - **Descrizione:**
    La classe offre il servizio di parsing del JSON inviato dal client che verrà trasformato, utilizzando il template di Moustache, in codice sorgente Java.
  - **Utilizzo:**
    Il servizio in questione, utilizzando il template pre-compilato dal midLoader, si occupa di renderizzare il JSON inviato dal client, convertire lo stream risultante in in JSON e rispedirlo al chiamante tramite callback.

4.1.5 SWEDesigner::Server::Controller::Services::UserServices

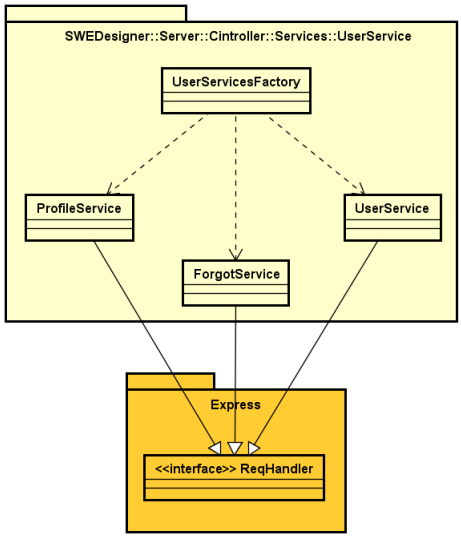

Figura 4: Diagramma dei packages SWEDesigner::Server::Controller::Services::UserServices

4.1.5.1 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

4.1.5.2 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

4.1.6 SWEDesigner::Server::Model

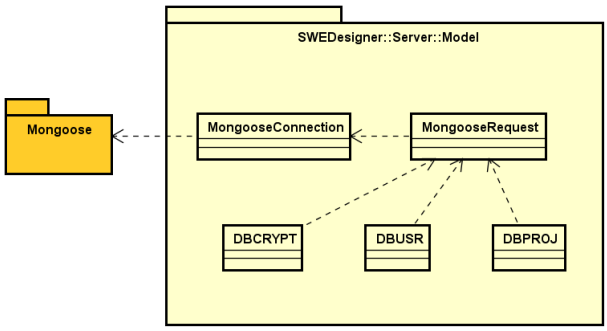

Figura 5: Diagramma dei packages SWEDesigner::Server::Model

4.1.6.1 Informazioni sul Package

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

4.1.6.2 Informazioni sulle Classi

- SWEDesigner::Server::Model::MongooseConnection
  - **Descrizione:**
    La classe stabilisce la connessione al database.
  - **Utilizzo:**
    La classe si connette al database passando per il modulo di Mongoose.
- SWEDesginer::Server::Model::mongooseRequest
  - **Descrizione:**
    La classe si occupa di gestire le interrogazioni al database, comprendendo il drop dello schema
  - **Utilizzo:**
    La classe si occupa di gestire tutte le query ritornando un errore o il valore atteso qualora questo sia presente nel 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.
- SWEDesigner::Server::Model::DBCRYPT
  - **Descrizione:**
    La classe gestisce il database dei dei parametri crittografici.

– **Utilizzo:**
La classe fornisce tutti gli strumenti per la lettura e la scrittura di una chiave crittografica.

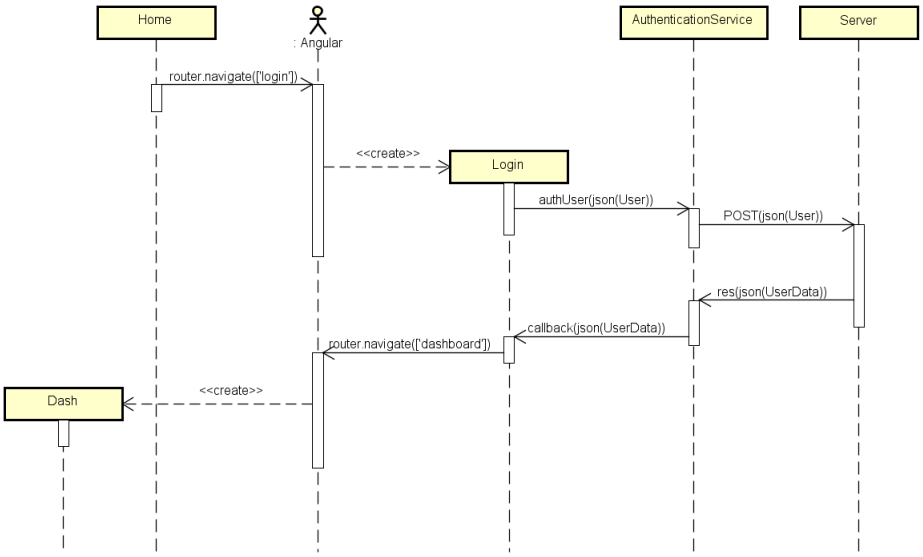

Figura 6: Esempio di funzionamento dell'applicazione lato client

5 Front-end

5.1 Descrizione packages e classi

5.1.1 SWEDesigner::Client

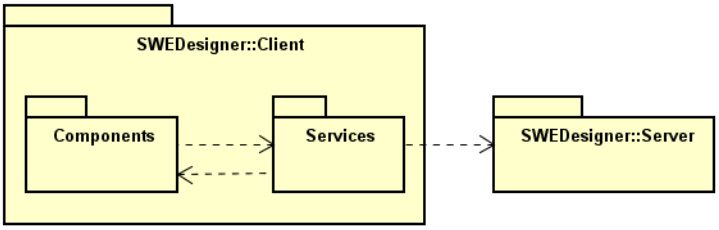

Figura 7: Diagramma dei packages SWEDesigner::Client

5.1.1.1 Informazioni sul Package

- **Descrizione:**
  Package che racchiude tutta la componente di Front-end scritta in TypeScript.
- **Padre:**
  SWEDesigner

- **Package contenuti:**
  - SWEDesigner::Client::Components;
  - SWEDesigner::Client::Services;
  - SWEDesigner::Server;

5.1.2 SWEDesigner::Client::Components

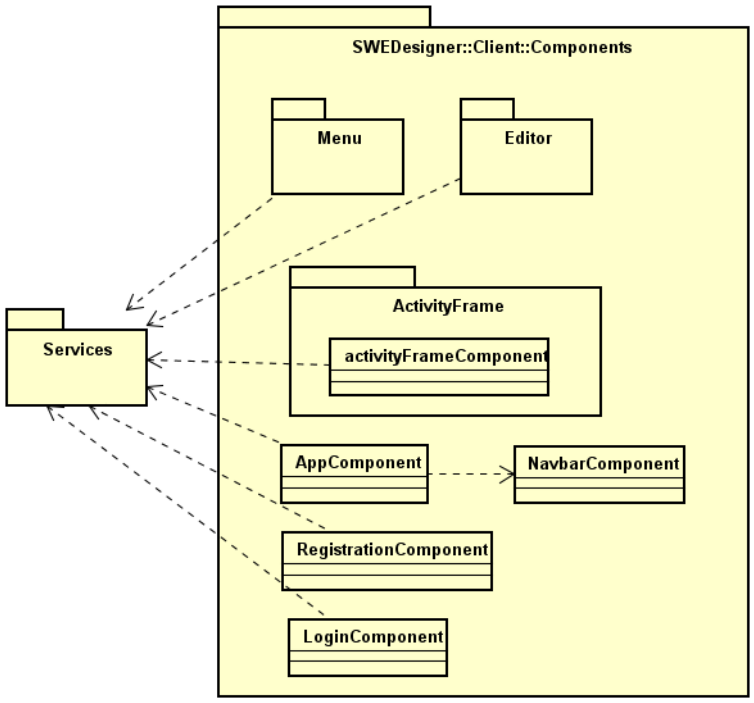

Figura 8: Diagramma dei packages SWEDesigner::Client::Components

5.1.2.1 Informazioni sul Package

- **Descrizione:**
  Questo package contiene tutti i components dell'applicazione
- **Padre:**
  SWEDesigner::Client
- **Package contenuti:**
  - SWEDesigner::Client::Components::Editor;
  - SWEDesigner::Client::Components::Menu;

– SWEDesigner::Client::Components::ActivityFrame;

5.1.2.2 Informazioni sulle Classi

- SWEDesigner::Client::Components::AppComponent
  - **Descrizione:**
    Il component descrive un contenitore per la barra di navigazione e le altre componenti dell'applicazione le quali sono istanziate dinamicamente all' interno del template http;
  - **Utilizzo:**
    AppComponent è il primo component che viene istanziato tramite bootsrap.
  - **Relazioni con altre classi:**
    - *OUT* SWEDesigner::Client::Services::editorService. Utilizza i servizi per effettuare azioni all'interno dell'editor.
    - *OUT* SWEDesigner::Client::Services::menuService. Utilizza i servizi per l'applicazione delle funzionalità del menu.
    - *OUT* SWEDesigner::Client::Services::NavbarComponent. Permette la navigazione all'interno dell'applicazione
- SWEDesigner::Client::Components::NavbarComponent
  - **Descrizione:**
    Questo component permette la navigazione all'interno dell'applicazione tramite links;
  - **Utilizzo:**
    NavbarComponent è istanziato per bootstrap subito dopo dell'AppComponent
  - **Relazioni con altre classi:**
    - *IN* SWEDesigner::Client::Component::AppComponent. Richiama il componente di navigazione.
- SWEDesigner::Client::Components::RegistrationComponent
  - **Descrizione:**
    È il componente che descrive la pagina di registrazione dell'applicazione, mette a disposizione dell'utente un form dove iserire le informazioni necessarie alla creazione di un nuovo account utente. Gestisce le operazioni e la logica applicativa per la registrazione servendosi dei metodi forniti dal servizio AuthenticationService;

  - **Utilizzo:**
    Questo componente viene instanziato dinamicamente dal servizio Router del framework Angular qunado viene richiesta la pagina di registrazione.
  - **Relazioni con altre classi:**
    * *OUT* SWEDesigner::Client::Services::accountService. Permette le operazioni di registrazione.

- SWEDesigner::Client::Components::LoginComponent
  - **Descrizione:**
    È il componente che descrive la pagina di login dell'applicazione, mette a disposizione dell'utente un form dove iserire username e password. Gestisce le operazioni e la logica applicativa per il login servendosi dei metodi forniti dal servizio AuthenticationService;
  - **Utilizzo:**
    Questo componente viene instanziato dinamicamente dal servizio Router del framework Angular qunado viene richiesta la pagina di login.
  - **Relazioni con altre classi:**
    * *OUT* SWEDesigner::Client::Services::accountService. Permette le operazioni di login.

5.1.3 SWEDesigner::Client::Components::Editor

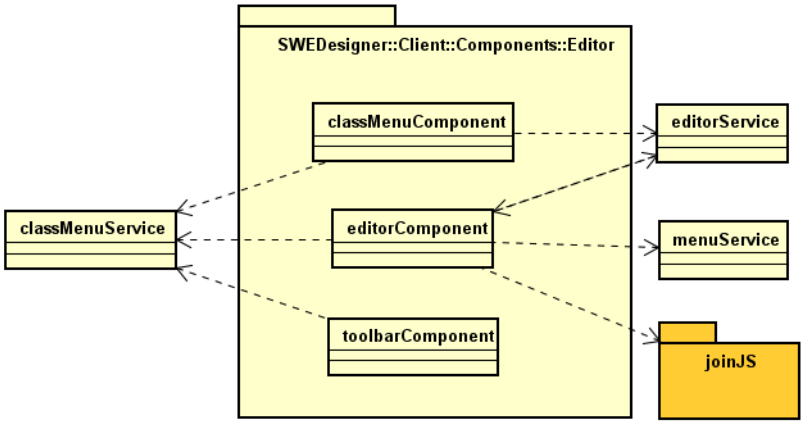

Figura 9: Diagramma dei packages SWEDesigner::Client::Components::Editor

5.1.3.1 Informazioni sul Package

- **Descrizione:**
  Il package contiene tutte le components riguardanti l'editor dei diagrammi.
- **Padre:**
  SWEDesigner::Client::Components

5.1.3.2 Informazioni sulle Classi

- SWEDesigner::Client::Components::Editor::ToolbarComponent
  - **Descrizione:**
    ToolbarComponent descrive il menu dal quale l'utente può selezionare gli strumenti per disegnare i diagrammi all'interno degli appositi frame. Si occupa delle operazioni e della parte logica, riguardante la costruzione dei diagrammi, servendosi dei metodi forniti delle API della libreria grafica;
  - **Utilizzo:**
    ToolbarComponent componente viene istanziato per bootstrap dopo che è stato istanziato il component AppComponent.
  - **Relazioni con altre classi:**
    - *OUT* SWEDesigner::Client::Services::classMenuService. Permette le operazioni di selezione dal menu degli strumenti.
- SWEDesigner::Client::Components::Editor::editorComponent
  - **Descrizione:**
    Component che contiene la rappresentazione grafica dei diagrammi disegnati dall'utente
  - **Utilizzo:**
    Questo componente viene instanziato dinamicamente dal servizio Router del framework Angular quando viene richiesta la pagina dell'editor diagrammi.
  - **Relazioni con altre classi:**
    - *OUT* SWEDesigner::Client::Services::classMenuService. Permette le operazioni modifica dei campi di un oggetto selezionato.
    - *OUT* SWEDesigner::Client::Services::menuService. Permette l'applicazione delle funzionalità fornite dal menu.
    - *OUT* SWEDesigner::Client::Services::editorService. Permette le operazioni di iterazione con il componente classMenuComponent.
    - *IN* SWEDesigner::Client::Services::editorService. Permette le operazioni di ricezione delle informazioni che vengono elaborate dal componente classMenuComponent.

    * *OUT* jointJS. Permette l'utilizzo delle funzionalità fornite dalla libreria grafica.

- SWEDesigner::Client::Components::Editor::classMenuComponent
  - **Descrizione:**
    Component che permette la modifica dei campi dati di un oggetto selezionato nell'editorComponent
  - **Utilizzo:**
    Component figlio di editorComponent viene visualizzato quando viene selezionato un elemento editabile nell'editorComponent.
  - **Relazioni con altre classi:**
    * *OUT* SWEDesigner::Client::Services::editorService. Permette le operazioni di comunicazione con il componente editorComponent.
    * *OUT* SWEDesigner::Client::Services::classMenuService. Permette le operazioni di iterazione con il componente editorComponent per visualizzare o meno il component.

5.1.4 SWEDesigner::Client::Components::Menu

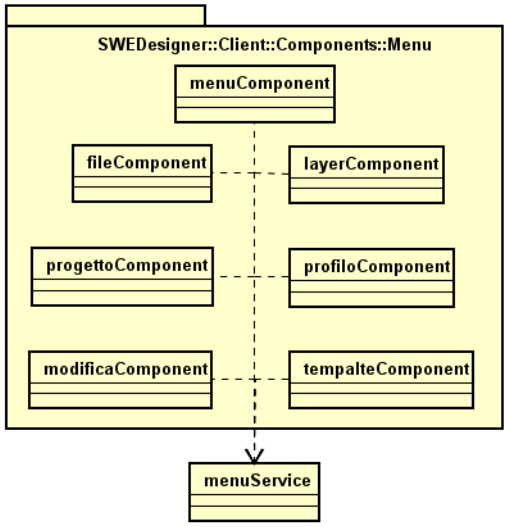

Figura 10: Diagramma dei packages SWEDesigner::Client::Components::Menu

5.1.4.1 Informazioni sul Package

– **Descrizione:**
Il package contiene tutti i components riguardanti la gestione delle funzionalirà fornite dal menu.

– **Padre:**
SWEDesigner::Client::Components

5.1.4.2 Informazioni sulle Classi

– SWEDesigner::Client::Component::Menu::menuComponent

* **Descrizione:**
Component che contiene l'insieme di funzionalità fornite all'utente per la gestione dei progetti, dei propri dati personali, e della rappresentazione dei grafici su cui sta lavorando.

* **Utilizzo:**
menuComponent viene istanziato per bootstrap dopo che è stato istanziato il component appComponent;

* **Relazione con altre classi:**

  · *OUT* SWEDesigner::Client::Services::menuService.

– SWEDesigner::Client::Component::Menu::fileComponent

* **Descrizione:**
Component che contiene l'insieme di funzionalità fornite all'utente per la gestione del progetto attualmente in uso.

* **Utilizzo:**
fileComponent viene istanziato per bootstrap dopo che è stato istanziato il component menuComponent;

* **Relazione con altre classi:**

  · *OUT* SWEDesigner::Client::Services::menuService. Permette le operazioni di salvataggio ed esportazione del progetto in uso.

– SWEDesigner::Client::Component::Menu::layerComponent

* **Descrizione:**
Component che contiene l'insieme di funzionalità fornite all'utente per la gestione dei layer del progetto in uso.

* **Utilizzo:**
layerComponent viene istanziato per bootstrap dopo che è stato istanziato il component menuComponent;

    * **Relazione con altre classi:**

        · *OUT* SWEDesigner::Client::Services::menuService. Permette le operazioni di gestione dei layer del progetto in uso.

- SWEDesigner::Client::Component::Menu::progettoComponent
    * **Descrizione:**
      Component che contiene l'insieme di funzionalità fornite all'utente per la gestione dei propri progetti salvati
    * **Utilizzo:**
      progettoComponent viene istanziato per bootstrap dopo che è stato istanziato il component menuComponent;
    * **Relazione con altre classi:**

        · *OUT* SWEDesigner::Client::Services::menuService. Permette le operazioni di gestione dei propri progetti.

- SWEDesigner::Client::Component::Menu::profiloComponent
    * **Descrizione:**
      Component che contiene l'insieme di funzionalità fornite all'utente per la gestione dei propri dati personali.
    * **Utilizzo:**
      profiloComponent viene istanziato per bootstrap dopo che è stato istanziato il component menuComponent;
    * **Relazione con altre classi:**

        · *OUT* SWEDesigner::Client::Services::menuService. Permette le operazioni di gestione dei propri dati personali.

- SWEDesigner::Client::Component::Menu::modificaComponent
    * **Descrizione:**
      Component che contiene l'insieme di funzionalità fornite all'utente per la modifica del progetto in uso, come ad esempio effettuare lo zoom, oppre eliminare o copiare un elemento selezionato.
    * **Utilizzo:**
      modificaComponent viene istanziato per bootstrap dopo che è stato istanziato il component menuComponent;
    * **Relazione con altre classi:**

- *OUT* SWEDesigner::Client::Services::menuService. Permette le operazioni di modifica sugli elementi del progetto in uso.

– SWEDesigner::Client::Component::Menu::templateomponent

* **Descrizione:**
  Component che contiene l'insieme di funzionalità fornite all'utente per l'importazione e gestione dei template.
* **Utilizzo:**
  templateComponent viene istanziato per bootstrap dopo che è stato istanziato il component menuComponent;
* **Relazione con altre classi:**

  - *OUT* SWEDesigner::Client::Services::menuService. Permette le operazioni di gestione ed importazione dei template.

5.1.5 SWEDesigner::Client::Components::ActivityFrame

5.1.5.1 Informazioni sul Package

– **Descrizione:**
Il package contiene i components riguardanti la gestione dell'activity frame, per la visione del flusso del programma.

– **Padre:**
SWEDesigner::Client::Components

5.1.5.2 Informazioni sulle Classi

– SWEDesigner::Client::Components::ActivityFrame::activityFrameComponent

* **Descrizione:**
  Component che descrive la struttura del frame dove l'utente può visualizzare l'activity frame che rappresenta il flusso logico del programma.
* **Utilizzo:**
  Questo component viene istanziato per bootstrap dopo l'istanziazione del component AppComponent.
* **Relazioni con altre classi:**

  - *OUT* SWEDesigner::Client::Services::activityFrameService. Permette le operazioni navigazione dell'activity frame.

5.1.6 SWEDesigner::Client::Service

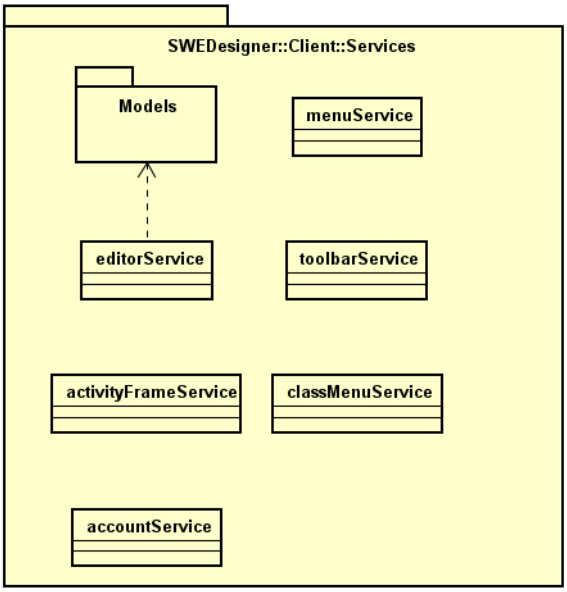

Figura 11: Diagramma dei packages SWEDesigner::Client::Service

5.1.6.1 Informazioni sul Package

- **Descrizione:**
  Il package contiene i servizi per le operazioni con la libreria grafica *jointJS* e con il server.
- **Padre:**
  SWEDesigner::Client
- **Package contenuti:**
  - SWEDesigner::Client::Services::Models;

5.1.6.2 Informazioni sulle Classi

- SWEDesigner::Client::Services::menuService
  - **Descrizione:**
    Classe che definisce i metodi per le operazioni fornite all'utente dal menu;
  - **Utilizzo:**
    É istaziata dal framework Angular e i suoi metodi sono utilizzati dal component menuComponent;

    - **Relazioni con altre classi:**
        - *IN* SWEDesigner::Client::Components::Menu::menuComponent
        - *IN* SWEDesigner::Client::Components::Menu::fileComponent. Fornisce i metodi per il salvataggio del file in uso e la generazione del codice;
        - *IN* SWEDesigner::Client::Components::Menu::layerComponent. Fornisce i metodi per la gestione dei layer;
        - *IN* SWEDesigner::Client::Components::Menu::progettoComponent. Fornisce i metodi per la gestione dei progetti;
        - *IN* SWEDesigner::Client::Components::Menu::profiloComponent. Fornisce i metodi per la gestione del profilo utente;
        - *IN* SWEDesigner::Client::Components::Menu::modificaComponent. Fornisce i metodi per la modifica degli elementi dei diagrammi;
        - *IN* SWEDesigner::Client::Components::Menu::templateComponent. Fornisce i metodi per la gestione dei template;
        - *IN* SWEDesigner::Client::Components::Editor::editorComponent. Fornisce i metodi per effettuare operazioni all'interno dell'editor dei diagrammi.

- SWEDesigner::Client::Services::editorService
    - **Descrizione:**
      Classe che definisce i metodi per le operazioni all'interno dei diagrammi e la comunicazione tra componenti e server;
    - **Utilizzo:**
      É istaziata dal framework Angular e i suoi metodi sono utilizzati dai component editorComponent e classMenuComponent;
    - **Relazioni con altre classi:**
        - *IN* SWEDesigner::Client::Components::editorComponent. Fornisce i metodi per effettuare operazioni all'interno dell'editor dei diagrammi;
        - *IN* SWEDesigner::Client::Components::classMenuComponent. Fornisce i metodi per effettuare operazioni su un elemento selezionato all'interno dell'editor dei diagrammi;
        - *IN* SWEDesigner::Client::Services::Models::Global. Fornisce i metodi per la storicizzazione dei dati in liste;

- SWEDesigner::Client::Services::toolbarService

* **Descrizione:**
  Classe che definisce i metodi per le operazioni di inserimento di nuovi elementi all'interno dell'editor di diagrammi;
* **Utilizzo:**
  É istaziata dal framework Angular e i suoi metodi sono utilizzati dal component editorComponent;
* **Relazioni con altre classi:**
  - *IN* SWEDesigner::Client::Components::Editod::editorComponent. Fornisce i metodi per inserire nuovi elementi all'interno dell'editor dei diagrammi;

– SWEDesigner::Client::Services::activityFrameService

* **Descrizione:**
  Classe che definisce i metodi per le operazioni di navigazione tra i metodi all'interno dell'activity frame;
* **Utilizzo:**
  É istaziata dal framework Angular e i suoi metodi sono utilizzati dal component activityFrameComponent;
* **Relazioni con altre classi:**
  - *IN* SWEDesigner::Client::Components::ActivityFrame::activityFrameComponent. Fornisce i metodi per la navigazione tra i vari metodi all'interno dell'activity frame;

– SWEDesigner::Client::Services::classMenuService

* **Descrizione:**
  Classe che definisce i metodi per le operazioni di modifica di un elemento selezionato all'interno del diagramma rappresentato
* **Utilizzo:**
  É istaziata dal framework Angular e i suoi metodi sono utilizzati dal component classMenuService;
* **Relazioni con altre classi:**
  - *IN* SWEDesigner::Client::Components::Editor::classMenuComponent. Fornisce i metodi per la modifica di un elemento selezionato all'interno del diagramma;
  - *IN* SWEDesigner::Client::Components::Editor::editorComponent. Fornisce i metodi di comunicazione e di passaggio di informazioni tra editorComponent e classMenuComponent per un elemento selezionato;

– SWEDesigner::Client::Services::accountService

* **Descrizione:**
  Classe che definisce i metodi di registrazione, login e recupero dati utente dal server
* **Utilizzo:**
  É istaziata dal framework Angular e i suoi metodi sono utilizzati dai component registrationComponent e loginComponent;
* **Relazioni con altre classi:**
  · *IN* SWEDesigner::Client::Components::registrationComponent. Fornisce i metodi per la registrazione di un nuovo utente e la comunicazione con il server;
  · *IN* SWEDesigner::Client::Components::loginComponent. Fornisce i metodi per il login di un utente registrato e la convalida dal server.

5.1.7 SWEDesigner::Client::Service::Models

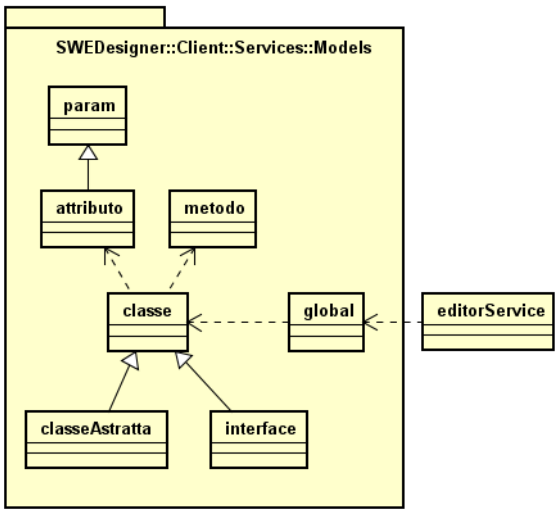

Figura 12: Diagramma dei packages SWEDesigner::Client::Service::Models

5.1.7.1 Informazioni sul Package

– **Descrizione:**
  Il package contiene moduli necessari a storicizzare i dati inseriti all'interno dei diagrammi.

– **Padre:**
SWEDesigner::Client::Service

5.1.7.2 Informazioni sulle Classi

– SWEDesigner::Client::Services::Modules::Param

* **Descrizione:**
Classe che definisce i metodi di settaggio e richiesta dei parametri nome e tipo;

* **Utilizzo:**
É istaziata dal framework Angular e i suoi metodi sono utilizzati dal model attributo

* **Relazioni con altre classi:**

· *IN* SWEDesigner::Client::Services::Models::Attributo. Utilizza i metodi di settaggio e richiesta dei parametri

– SWEDesigner::Client::Services::Modules::attributo

* **Descrizione:**
Classe derivata da Param che definisce i metodi di settaggio e richiesta dei parametri di visibilità;

* **Utilizzo:**
É istaziata dal framework Angular e i suoi metodi sono utilizzati dal model classe

* **Relazioni con altre classi:**

· *IN* SWEDesigner::Client::Services::Models::classe. Utilizza i metodi di settaggio e richiesta dei parametri e delle visibilità

– SWEDesigner::Client::Services::Modules::metodo

* **Descrizione:**
Classe che definisce i metodi di settaggio e richiesta dei metodi definiti all'interno dei diagrammi

* **Utilizzo:**
É istaziata dal framework Angular e i suoi metodi sono utilizzati dal model classe

* **Relazioni con altre classi:**

· *IN* SWEDesigner::Client::Services::Models::classe. Utilizza i metodi di settaggio e richiesta dei metodi

– SWEDesigner::Client::Services::Modules::classe

  - **Descrizione:**
    Classe che definisce i metodi di settaggio e richiesta di tutti gli elementi che sono contenuti in una classe. Contiene un array di metodi, con le relative rappresentazioni grafiche dei metodi implementati, e un array di attributi, oltre ai campi utili all'identificazione della classe
  - **Utilizzo:**
    É istaziata dal framework Angular e i suoi metodi sono utilizzati dal model global
  - **Relazioni con altre classi:**
    - *IN* SWEDesigner::Client::Services::Models::global. Utilizza i metodi di settaggio e richiesta delle classi;

- SWEDesigner::Client::Services::Modules::gobal
  - **Descrizione:**
    Classe che definisce i metodi di settaggio e richiesta di tutte le classi contenenti nel diagramma delle classi;
  - **Utilizzo:**
    É istaziata dal framework Angular e i suoi metodi sono utilizzati dal servizio editorService;
  - **Relazioni con altre classi:**
    - *IN* SWEDesigner::Client::Services::editorService. Utilizza i metodi di settaggio e richiesta delle classi storicizzate;

- SWEDesigner::Client::Services::Modules::classeAstratta
  - **Descrizione:**
    Classe derivata da classe che definisce i metodi di settaggio e richiesta dei parametri di una classe astratta;
  - **Utilizzo:**
    É istaziata dal framework Angular e i suoi metodi sono utilizzati dal model global
  - **Relazioni con altre classi:**
    - *IN* SWEDesigner::Client::Services::Models::global. Utilizza i metodi di settaggio e richiesta delle classi;

- SWEDesigner::Client::Services::Modules::interface
  - **Descrizione:**
    Rappresentazione di un'interfaccia di Java che definisce i metodi di settaggio e richiesta dei parametri di una interface;

* **Utilizzo:**
  É istaziata dal framework Angular e i suoi metodi sono utilizzati dal model global
* **Relazioni con altre classi:**
  - *IN* SWEDesigner::Client::Services::Models::global. Utilizza i metodi di settaggio e richiesta delle classi;

6 Tracciamento

In questa sezione vengono tracciate le corrispondenze tra componenti dell'architettura e i requisiti.

- Per ogni componente principale vengono mostrati i requisiti che ne motivano l'esisteza
- Per ogni requisito principale vengono mostrate le componenti principali che servono per soddisfarlo

6.1 Tracciamento componenti - requisiti

Componente	Requisito
SWEDesigner::Client::Components::AppComponent	R0F6
SWEDesigner::Client::Components::DashComponents	R1F3 R0F5 R0F6
SWEDesigner::Client::Components::DashComponents::DashComponent	R0F6
SWEDesigner::Client::Components::DashComponents::ProfileComponent	R1F3 R1F3.1 R1F3.2 R1F3.3 R1F13 R0F5 R0F5.1 R0F5.1.1 R0F5.1.2 R0F5.2 R0F5.3
SWEDesigner::Client::Components::EditorComponents	R0F6 R0F6.1 R0F6.2 R0F6.3 R0F6.4
SWEDesigner::Client::Components::EditorComponents::ActivityFrameComponent	R0F6.4
SWEDesigner::Client::Components::EditorComponents::BreadcrumbComponent	R0F6.4.1
SWEDesigner::Client::Components::EditorComponents::ClassFrameComponent	R0F6.3.1 R0F6.2.1

Componente	Requisito
SWEDesigner::Client::Components::EditorComponents::EditorComponent	R0F6.3
SWEDesigner::Client::Components::EditorComponents::MenuComponent	R0F6.1.1 R0F6.1.1.1 R0F6.1.1.2 R0F6.1.1.3 R0F6.1.1.4 R0F6.2
SWEDesigner::Client::Components::EditorComponents::MethodComponent	R0F6.3.5 R0F6.2.2
SWEDesigner::Client::Components::EditorComponents::ToolboxComponent	R0F6.1
SWEDesigner::Client::Components::LoginComponent	R0F2 R0F2.1 R0F2.2
SWEDesigner::Client::Components::RegistrationComponent	R0F1 R0F1.1 R0F1.2 R0F1.3 R0F1.4
SWEDesigner::Client::Services::ProjectServices::ExportServices	R0F6.1.1.3
SWEDesigner::Client::Services::ProjectServices::GeneratorServices	R0F6.1.1.4
SWEDesigner::Client::Services::ProjectServices::ProjManagServices	R1F14
SWEDesigner::Client::Services::UserServices	R0F1 R0F2
SWEDesigner::Client::Services::UserServices::AuthenticationService	R0F1 R0F1.1 R0F1.2 R0F1.3 R0F1.4
SWEDesigner::Client::Services::UserServices::ValidationServices	R0F2 R0F2.1 R0F2.2
SWEDesigner::Server::Controller::Middleware::Encrypt	R0F6.1.1.3 R0F6.1.1.4

Componente	Requisito
SWEDesigner::Server::Controller::Middleware::ErrHandler	R0F1.5 R0F1.6 R0F1.7 R0F2.3 R1F3.4 R1F3.5
SWEDesigner::Server::Controller::Middleware::Parse	R0F6.1.1.4
SWEDesigner::Server::Controller::Middleware::Profile	R0F1 R0F1.1 R0F1.2 R0F1.3 R0F2 R0F2.1 R0F2.2
SWEDesigner::Server::Controller::Services::EncryptServices	R0F6.1.1.3 R0F6.1.1.4
SWEDesigner::Server::Controller::Services::JavaGenService	R0F6.1.1.4
SWEDesigner::Server::Controller::Services::JavaGenService::DownloadService	R0F6.1.1.3 R0F6.1.1.4
SWEDesigner::Server::Controller::Services::JavaGenService::JavaGenFactory	R0F6.1.1.4
SWEDesigner::Server::Controller::Services::JavaGenService::ParseService	R0F6.1.1.4
SWEDesigner::Server::Controller::Services::UserServices	R0F1 R0F2
SWEDesigner::Server::Controller::Services::UserServices::ForgotService	R1F14
SWEDesigner::Server::Controller::Services::UserServices::ProfileService	R0F1
SWEDesigner::Server::Controller::Services::UserServices::UserService	R1f3 R1F3.1 R1F3.2 R1F3.3 R1F13

Componente	Requisito
SWEDesigner::Server::Controller::Services::UserServices::UserServicesFactory	R0F1 R0F1.1 R0F1.2 R0F1.3 R0F2 R0F2.1 R0F2.2
SWEDesigner::Server::Model::DBPROJ	R0F5 R0F5.1 R0F5.2 R0F5.3 R0F6.1.1.1 R0F6.1.1.5 R2F6.1.4.1
SWEDesigner::Server::Model::DBUSR	R0F1 R0F2
SWEDesigner::Server::Model::MongooseConnection	R0F5.2 R0F5.3 R0F6.1.1.1 R1F6.1.1.5 R1F6.1.3.2 R2F6.1.4.1 R2F6.1.4.3

Tabella 2: Tracciamento componenti - requisiti

6.2 Tracciamento requisiti - componenti

Requisito	Componente
R0F1	SWEDesigner::Client::Components::RegistrationComponent SWEDesigner::Client::Services::UserServices SWEDesigner::Client::Services::UserServices::AuthenticationService SWEDesigner::Server::Controller::Middleware::Profile SWEDesigner::Server::Controller::Services::UserServices SWEDesigner::Server::Controller::Services::UserServices::ProfileService SWEDesigner::Server::Controller::Services::UserServices::UserServicesFactory SWEDesigner::Server::Model::DBUSR

Requisito	Componente
R0F1.1	SWEDesigner::Client::Components::RegistrationComponent SWEDesigner::Server::Controller::Middleware::Profile SWEDesigner::Client::Services::UserServices::AuthenticationService SWEDesigner::Server::Controller::Services::UserServices::UserServicesFactory
R0F1.2	SWEDesigner::Client::Components::RegistrationComponent SWEDesigner::Client::Services::UserServices::AuthenticationService SWEDesigner::Server::Controller::Middleware::Profile SWEDesigner::Server::Controller::Services::UserServices::UserServicesFactory
R0F1.3	SWEDesigner::Client::Components::RegistrationComponent SWEDesigner::Client::Services::UserServices::AuthenticationService SWEDesigner::Server::Controller::Middleware::Profile SWEDesigner::Server::Controller::Services::UserServices::UserServicesFactory
R0F1.4	SWEDesigner::Client::Components::RegistrationComponent SWEDesigner::Client::Services::UserServices::AuthenticationService
R0F1.5	SWEDesigner::Server::Controller::Middleware::ErrHandler
R0F1.6	SWEDesigner::Server::Controller::Middleware::ErrHandler
R0F1.7	SWEDesigner::Server::Controller::Middleware::ErrHandler
R0F2	SWEDesigner::Client::Components::LoginComponent SWEDesigner::Client::Services::UserServices SWEDesigner::Server::Controller::Middleware::Profile SWEDesigner::Client::Services::UserServices::ValidationServices SWEDesigner::Server::Controller::Services::UserServices SWEDesigner::Server::Controller::Services::UserServices::UserServicesFactory SWEDesigner::Server::Model::DBUSR
R0F2.1	SWEDesigner::Client::Components::LoginComponent SWEDesigner::Client::Services::UserServices::ValidationServices SWEDesigner::Server::Controller::Middleware::Profile SWEDesigner::Server::Controller::Services::UserServices::UserServicesFactory
R0F2.2	SWEDesigner::Client::Components::LoginComponent SWEDesigner::Client::Services::UserServices::ValidationServices SWEDesigner::Server::Controller::Middleware::Profile SWEDesigner::Server::Controller::Services::UserServices::UserServicesFactory
R0F2.3	SWEDesigner::Server::Controller::Middleware::ErrHandler
R0F5	SWEDesigner::Client::Components::DashComponents SWEDesigner::Client::Components::DashComponents::ProjectComponent SWEDesigner::Server::Model::DBPROJ

Requisito	Componente
R0F5.1	SWEDesigner::Client::Components::DashComponents::ProjectComponent SWEDesigner::Server::Model::DBPROJ
R0F5.1.1	SWEDesigner::Client::Components::DashComponents::ProjectComponent
R0F5.1.2	SWEDesigner::Client::Components::DashComponents::ProjectComponent
R0F5.2	SWEDesigner::Client::Components::DashComponents::ProjectComponent SWEDesigner::Server::Model::DBPROJ SWEDesigner::Server::Model::MongooseConnection
R0F5.3	SWEDesigner::Client::Components::DashComponents::ProjectComponent SWEDesigner::Server::Model::DBPROJ SWEDesigner::Server::Model::MongooseConnection
R0F6	SWEDesigner::Client::Components::AppComponent SWEDesigner::Client::Components::DashComponents SWEDesigner::Client::Components::DashComponents::DashComponent SWEDesigner::Client::Components::EditorComponents
R0F6.1	SWEDesigner::Client::Components::EditorComponents SWEDesigner::Client::Components::EditorComponents::ToolboxComponent
R0F6.1.1	SWEDesigner::Client::Components::EditorComponents::MenuComponent
R0F6.1.1.1	SWEDesigner::Client::Components::EditorComponents::MenuComponent SWEDesigner::Server::Model::DBPROJ SWEDesigner::Server::Model::MongooseConnection
R0F6.1.1.2	SWEDesigner::Client::Components::EditorComponents::MenuComponent
R0F6.1.1.3	SWEDesigner::Client::Components::EditorComponents::MenuComponent SWEDesigner::Client::Services::ProjectServices::ExportServices SWEDesigner::Server::Controller::Middleware::Encrypt SWEDesigner::Server::Controller::Services::EncryptServices SWEDesigner::Server::Controller::Services::JavaGenService::DownloadService
R0F6.1.1.4	SWEDesigner::Client::Components::EditorComponents::MenuComponent SWEDesigner::Client::Services::ProjectServices::GeneratorServices SWEDesigner::Server::Controller::Middleware::Encrypt SWEDesigner::Server::Controller::Middleware::Parse SWEDesigner::Server::Controller::Services::JavaGenService SWEDesigner::Server::Controller::Services::EncryptServices SWEDesigner::Server::Controller::Services::JavaGenService::DownloadService SWEDesigner::Server::Controller::Services::JavaGenService::JavaGenFactory SWEDesigner::Server::Controller::Services::JavaGenService::ParseService

Requisito	Componente
R0F6.1.1.5	SWEDesigner::Server::Model::DBPROJ
R0F6.2	SWEDesigner::Client::Components::EditorComponents SWEDesigner::Client::Components::EditorComponents::MenuComponent
R0F6.2.1	SWEDesigner::Client::Components::EditorComponents::ClassFrameComponent
R0F6.2.2	SWEDesigner::Client::Components::EditorComponents::MethodComponent
R0F6.3	SWEDesigner::Client::Components::EditorComponents SWEDesigner::Client::Components::EditorComponents::EditorComponent
R0F6.3.1	SWEDesigner::Client::Components::EditorComponents::ClassFrameComponent
R0F6.3.5	SWEDesigner::Client::Components::EditorComponents::MethodComponent
R0F6.4	SWEDesigner::Client::Components::EditorComponents SWEDesigner::Client::Components::EditorComponents::ActivityFrameComponent
R0F6.4.1	SWEDesigner::Client::Components::EditorComponents::BreadcrumbComponent
R1F13	SWEDesigner::Client::Components::DashComponents::ProfileComponent SWEDesigner::Server::Controller::Services::UserServices::UserService
R1F14	SWEDesigner::Client::Services::ProjectServices::ProjManagServices SWEDesigner::Server::Controller::Services::UserServices::ForgotService
R1F3	SWEDesigner::Client::Components::DashComponents SWEDesigner::Client::Components::DashComponents::ProfileComponent SWEDesigner::Server::Controller::Services::UserServices::UserService
R1F3.1	SWEDesigner::Client::Components::DashComponents::ProfileComponent SWEDesigner::Server::Controller::Services::UserServices::UserService
R1F3.2	SWEDesigner::Client::Components::DashComponents::ProfileComponent SWEDesigner::Server::Controller::Services::UserServices::UserService
R1F3.3	SWEDesigner::Client::Components::DashComponents::ProfileComponent SWEDesigner::Server::Controller::Services::UserServices::UserService
R1F3.4	SWEDesigner::Server::Controller::Middleware::ErrHandler
R1F3.5	SWEDesigner::Server::Controller::Middleware::ErrHandler
R1F6.1.1.5	SWEDesigner::Server::Model::MongooseConnection
R1F6.1.3.2	SWEDesigner::Server::Model::MongooseConnection
R2F6.1.4.1	SWEDesigner::Server::Model::DBPROJ SWEDesigner::Server::Model::MongooseConnection

Requisito	Componente
R2F6.1.4.3	SWEDesigner::Server::Model::MongooseConnection

Tabella 3: Tracciamento requisiti - componenti

A Descrizione *Design Pattern*$_G$

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*$_G$ implementati nella costruzione dell'architettura di alto livello, divisi per categoria di applicazione:

A.1 *Design Pattern*$_G$ Architetturali

A.1.1 MVVM

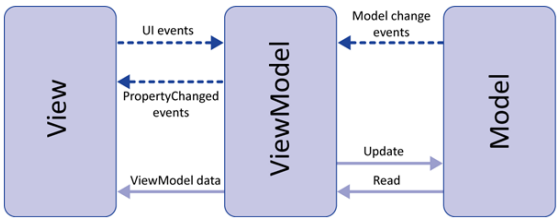

Figura 13: Diagramma del *Design Pattern*$_G$ MVVM

- **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 *Database*$_G$, 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 (UX)*$_G$ di scrivere il codice *GUI*$_G$, possono utilizzare il linguaggio di marcatura dei *framework*$_G$ 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.

- **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

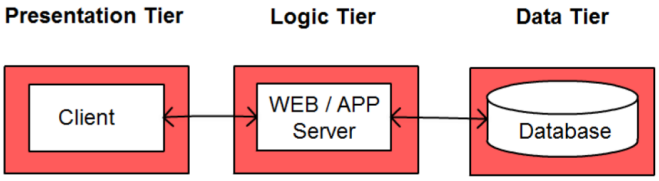

Figura 14: Diagramma del *Design Pattern*$_G$ Three-Tier

- **Scopo**: tale *Design Pattern*$_G$ 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*$_G$ 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*$_G$ Creazionali

A.2.1 Factory Method

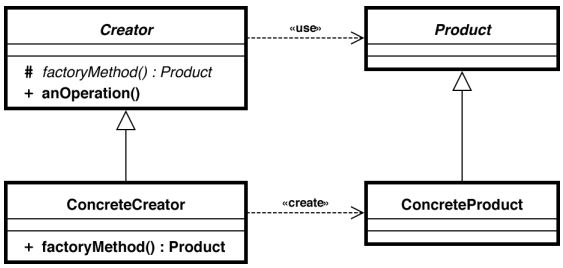

Figura 15: Diagramma del *Design Pattern*$_G$ Factory method

- **Scopo**: il *Design Pattern*$_G$ 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*$_G$ 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*$_G$ Strutturali

A.3.1 Decorator

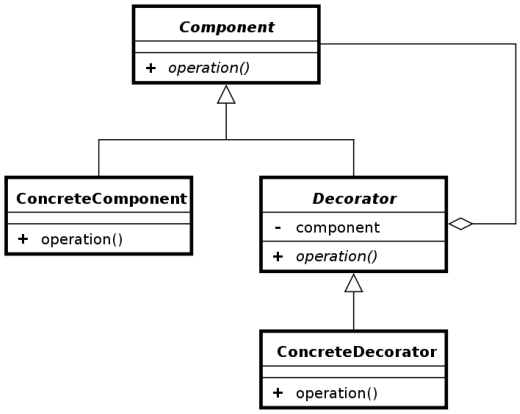

Figura 16: Diagramma del *Design Pattern*$_G$ Decorator

- **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

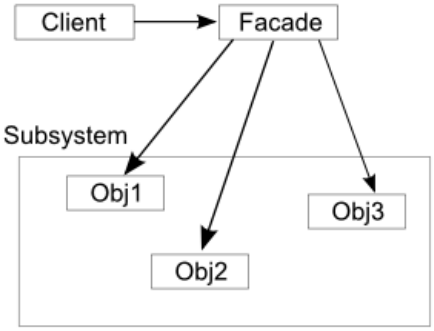

Figura 17: Diagramma del *Design Pattern*$_G$ Facade

- **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*$_G$ 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*$_G$ Comportamentali

A.4.1 Dependency Injection

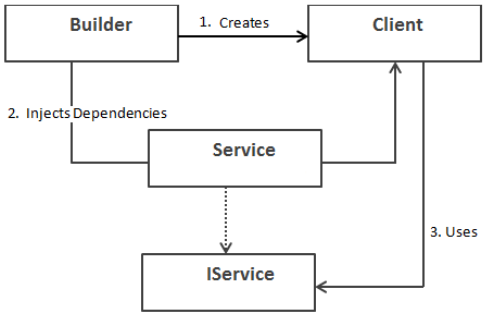

Figura 18: Diagramma del *Design Pattern*$_G$ Dependency Injection

- **Scopo**: Il Dependency Injection è un *Design Pattern*$_G$ che permette la separazione del comportamento degli oggetti dalla loro dipendenze. Invece di istanziare le *classi*$_G$ 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*$_G$ che implementano una data dipendenza su richiesta.

  Il dependent object dichiara da quali componenti dipende. L'injector decide quali *classi*$_G$ 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

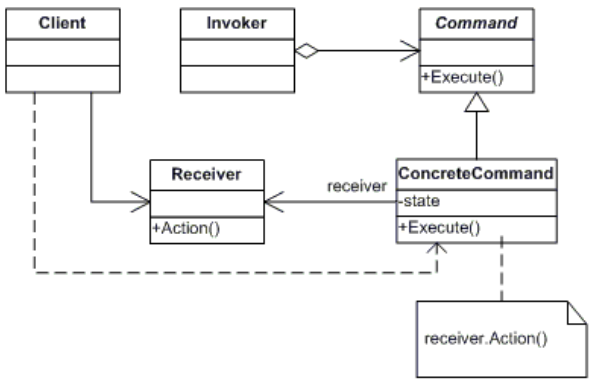

Figura 19: Diagramma del *Design Pattern*$_G$ Command

- **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.