Università degli Studi di Padova

KaleidosCode

SWEDesigner

Software per diagrammi UML

Specifica tecnica v1.0.0

Informazioni sul documento

Versione	1.0.0
Data Redazione	01/05/2017
Redazione	Bonolo Marco
	Pace Giulio
	Pezzuto Francesco
	Sanna Giovanni
Verifica	Sovilla Matteo
Approvazione	Bonato Enrico
Uso	Esterno
Distribuzione	*Prof. Vardanega Tullio*
	Prof. Cardin Riccardo
	Zucchetti s.p.a.

kaleidos.codec6@gmail.com

Diario delle Modifiche

Versione	Data	Autore	Descrizione
0.0.4	02/05/2017	Pezzuto Francesco	Sistemata Introduzione; Aggiunte sezioni Tecnologie Utilizzate, Architettura generale, Design pattern utilizzati
0.0.1	01/05/2017	Pace Giulio	Creazione scheletro del documento e stesura della sezione Introduzione

Indice

Elenco delle tabelle

Elenco delle figure

1 Introduzione

1.1 Scopo del documento

Con il presente documento si intende definire la progettazione ad alto livello del progetto *SWEDesigner*.
Verrà presentata innanzi tutto l'architettura generale secondo la quale verranno organizzate le componenti software. Successivamente verranno descritti i Design pattern$_G$ utilizzati.

1.2 Scopo del prodotto

Lo scopo del progetto è la realizzazione di un software di costruzione di diagrammi UML$_G$ con la relativa generazione di codice Java$_G$ e Javascript$_G$ utilizzando tecnologie web. Il prodotto deve essere conforme ai vincoli qualitativi richiesti dal committente.

1.3 Glossario

Al fine di evitare ogni ambiguità di linguaggio e massimizzare la comprensione dei documenti i termini tecnici, di dominio, gli acronimi e le parole che necessitano di essere chiarite sono riportate nel documento *Glossario v2.0.0*.
La prima occorrenza di ciascuno di questi vocaboli è marcata da una "G" maiuscola in pedice.

1.4 Riferimenti utili

1.4.1 Riferimenti normativi

- **Capitolato$_G$ d'appalto**:
  http://www.math.unipd.it/~tullio/IS-1/2016/Progetto/C6.pdf (09/03/2017);
- **Norme di progetto**: *Norme di progetto v2.0.0*;
- **Analisi dei requisiti**: *Analisi dei requisiti v2.0.0*;
- **Verbali esterni**:
  - Verbale incontro con *Zucchetti s.p.a.* in data 05/05/2017.

1.4.2 Riferimenti informativi

- **Slide dell'insegnamento di Ingegneria del Software 1° semestre**:
  - Design pattern strutturali:
    http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E04.pdf (02/05/2017);
  - Design pattern creazionali:
    http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E05.pdf (02/05/2017);
  - Design pattern comportamentali:
    http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E06.pdf (02/05/2017);

  - Design pattern architetturali:
    http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E07.pdf (02/05/2017),
    http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E08.pdf (02/05/2017);
  - Stili architetturali:
    http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E09.pdf (02/05/2017);

- **Design Patterns: Elements of reusable object-oriented software**
  E. Gamma, R. Helm, R. Johnson, J. Vlissides - 1st Edition (2002)
  - Capitolo 3: Creational patterns;
  - Capitolo 4: Structural patterns;
  - Capitolo 5: Behavioral patterns.

2 Tecnologie utilizzate

2.1 HTML5

Linguaggio per la strutturazione delle pagine web, come richiesto dal Proponente.

Principali vantaggi

- Possibilità di gestire immagini (canvas) e audio direttamente attraverso Javascript;
- Linguaggio ben documentato e quindi di ridotta complessità;
- Il suo corretto utilizzo permette di separare struttura e contenuti delle pagine web, dalla loro presentazione e comportamento che vengono realizzate con altri linguaggi, aumentando quindi la manutenibilità del prodotto.

Principali svantaggi

- Linguaggio non ancora pienamente supportato da tutti i browser$_G$.

2.2 CSS

Linguaggio per la formattazione e presentazione delle pagine HTML$_G$, come richiesto dal Proponente.

Principali vantaggi

- Il suo corretto utilizzo permette di separare totalmente la presentazione delle pagine HTML;
- Diminuisce i tempi di sviluppo e restyling di un sito, aumentandone quindi la manutenibilità;
- Consente di produrre pagine più leggere, riducendo i tempi di attesa per gli utenti;
- Il suo corretto utilizzo consente di aumentare l'accessibilità di un sito a screen-reader$_G$, browser testuali e dispositivi alternativi.

Principali svantaggi

- La versione 3 del linguaggio non è ancora pienamente supportata da tutti i browser.

2.3 Javascript

Linguaggio utilizzato per la realizzazione del comportamento delle pagine HTML, come richiesto dal Proponente.

Principali vantaggi

- Il codice Javascript viene eseguito dal browser dell'utente; di conseguenza il server non è sfruttato più del dovuto;
- Rende dinamiche le pagine web, permettendo al sito di reagire ad eventi generati dall'interazione dell'utente;
- Rende possibile interagire con il DOM$_G$.

Principali svantaggi

- Il codice è visibile e può essere letto da chiunque.

2.4 JointJS

Libreria Javascript scelta per la creazione dell'editor dei diagrammi UML.
(https://www.jointjs.com/opensource - 02/05/2017)

Principali vantaggi

- Elementi grafici di diagrammi UML pronti all'uso;
- Elementi e collegamenti interattivi;
- Serializzazione/de-serializzazione da/a formato JSON;
- Supporto a Node.js$_G$.

Principali svantaggi

- La versione open-source utilizzata è dipendente da altre librerie (jQuery, Lodash, Backbone), rendendo le ulteriori scelte tecnologiche obbligate all'utilizzo di queste ultime.

2.5 jQuery

Libreria Javascript utilizzata da JointJS utile allo sviluppo di script$_G$ lato client.
(https://jquery.com/ - 02/05/2017)

Principali vantaggi

- Facilita la manipolazione del DOM;
- Facilita lo sviluppo di comunicazioni asincrone tra client e server utilizzanti AJAX;
- Facilita la realizzazione di animazioni a livello base;
- Buon supporto da parte della community.

Principali svantaggi

- Non tutta la libreria è sviluppata rispettando uno standard comune, rendendo eventualmente necessario manipolarne il codice per i propri bisogni;
- Può rallentare un sito nel caso di manipolazioni multiple simultanee del DOM.

2.6 Lodash

Libreria Javascript utilizzata da JointJS utile per svolgere operazioni di base all'interno di script.
(https://lodash.com/ - 02/05/2017)

Principali vantaggi

- Fornisce molte funzionalità per cercare di supportare il maggior numero di ambienti e requisiti;
- Buon supporto da parte della community.

2.7 Backbone.js

Framework$_G$ Javascript utilizzato da JointJS utile per fornire una struttura ad applicazioni web rendendo disponibili modelli con binding chiave-valore ed eventi personalizzati, collezioni con una API$_G$ contenente funzioni enumerabili, viste con gestione degli eventi dichiarativa ed un'interfaccia JSON RESTful.
(http://backbonejs.org/ - 02/05/2017)

Principali vantaggi

- Compatto e versatile;
- Contiene solo le componenti base necessarie a strutturare una web app secondo il pattern MVC;
- Ha una buona documentazione, inoltre è presente una versione commentata del codice sorgente dove è quindi spiegato come lavora nel dettaglio;
- Supporta plugins$_G$ di terze parti.

Principali svantaggi

- Non supporta il data binding$_G$ bidirezionale;
- È difficile eseguire test di unità sulle views scrivendo poco codice per mocking$_G$.

2.8 Node.js

Runtime Javascript open-source scelto per sviluppare la parte server di *SWEDesigner* come richiesto dal Proponente (Requisito R0V1); utilizza un modello I/O non bloccante ad eventi asincroni ed è costruito sul motore Javascript v8; non è multi-threaded$_G$, ma funziona in un singolo thread con il concetto di callback, inoltre esegue loop basato su eventi a singolo thread così da rendere non bloccanti tutte le esecuzioni.
(https://nodejs.org/it/ - 02/05/2017)

Principali vantaggi

- I/O ad eventi asincroni aiutano la gestione di richieste simultanee;
- Condivide la stessa porzione di codice con entrambi i lati client e server;
- Community molto attiva con molto codice condiviso via GitHub$_G$, ecc.

Principali svantaggi

- Rende complessa la gestione di database$_G$ relazionali.

2.9 JSON

JavaScript Object Notation, è il formato scelto per l'interscambio di dati tra client e server; è basato su Javascript, inoltre viene utilizzato in AJAX come alternativa a XML$_G$. JSON è basato su due strutture:

- Un insieme di coppie nome/valore; in diversi linguaggi questo è realizzato come un oggetto, uno struct, una tabella hash, un array associativo, ecc.;
- Un elenco ordinato di valori; nella maggior parte dei linguaggi questo è realizzato con un array, un vettore, un elenco, ecc.;

Principali vantaggi

- Leggero e supportato bene da tutti i browser;
- Formato conciso grazie al suo approccio basato su coppia nome/valore;
- Modo completamente automatizzato di serializzare/de-serializzare gli oggetti Javascript che richiede poco sviluppo di codice;
- API semplice;
- Supportato da molti toolkit di AJAX e librerie Javascript.

Principali svantaggi

- Nessun supporto a namespace che porta ad una scarsa estensibilità;
- Nessun sostegno per la definizione della grammatica formale; di conseguenza i contratti di interfaccia sono difficili da comunicare e far rispettare;
- Supporta strumenti di sviluppo limitati.

2.10 AJAX

Asynchronous Javascript And Xml, è la tecnica scelta per lo sviluppo della comunicazione dei client verso il server.

Principali vantaggi

- Migliora l'esperienza utente, "nascondendo" l'aggiornamento della pagina web;
- Riduce l'uso di banda e velocizza i caricamenti;
- È compatibile con molti linguaggi ed è supportato da molti browser.

Principali svantaggi

- Può rendere difficile il debug$_G$ della pagina web sulla quale è utilizzato poiché aumenta la dimensione del suo codice sorgente;
- Può incrementare il carico sul server web nel caso in cui si utilizzi per aggiornare troppo frequentemente una pagina.

2.11 RequireJS

Loader di moduli e file Javascript ottimizzato per l'uso in-browser ma anche per altri ambienti Javascript come Node.js.
(http://requirejs.org/ - 02/05/2017)

Principali vantaggi

- Buon supporto alla separazione del codice;
- Supporto a plugins;
- Può caricare moduli in modo asincrono "su richiesta".

Principali svantaggi

- Strumento difficile da usare efficacemente.

2.12 MySQL

DBMS$_G$ SQL relazionale scelta per lo sviluppo della base dati del sistema.
(https://www.mysql.com/ - 02/05/2017)

Principali vantaggi

- Tanto famoso quanto solido per sviluppare basi di dati;
- È progettato con in mente il web, il cloud e big data$_G$;
- Supporta moli di dati che possono essere anche molto grandi ed è veloce.

Principali svantaggi

- Non supporta alcune tipologie di join;
- Non supporta la possibilità di fare sub-query senza rieseguirle ogni volta.

3 Architettura generale

SWEDesigner è realizzato utilizzando un'architettura client-server, in particolare:

- Il **client** corrisponde alla parte dell'applicativo che funzionerà nel browser dell'utente;
- Il **server** avrà il compito di fornire la pagina dell'applicativo al client e ne gestirà le richieste ricevute riguardanti la generazione del codice sorgente o le attività "bubble" da inserire nell'editor.

3.1 Architettura client

Il client (parte front-end$_G$) è una Single Page Application (SPA$_G$) scritta con i linguaggi HTML5, CSS$_G$ e Javascript.
La sua architettura è costruita utilizzando il framework Backbone.js che offre un'architettura di tipo Model-View ed è quindi principalmente suddivisa nei seguenti moduli:

- **Model**: organizza la logica alla base dei diagrammi dell'editor.
- **View**: gestisce l'interfaccia grafica dell'editor e, seguendo la struttura definita da Backbone.js, "contiene" la componente controller per la gestione degli eventi;

3.1.1 Diagrammi editabili

In ogni diagramma creabile all'interno dell'applicazione è offerto solamente un sottoinsieme del totale dei formalismi definiti dal linguaggio UML standard. Si possono individuare quattro tipi di diagrammi:

- Diagramma dei package$_G$;
- Diagramma delle classi;
- Diagramma delle attività;
- Diagramma delle bubble.

Il diagramma dei package è logicamente correlato con il diagramma delle classi. Per ogni elemento (package o classe) all'interno di questi diagrammi è possibile assegnare un livello di importanza attraverso il quale si può "filtrare" gli oggetti a schermo visualizzabili nell'editor.
Per la corretta generazione del codice, nei diagrammi delle attività è previsto che l'utente approfondisca il loro livello di astrazione fino ad arrivare ad un diagramma costituito solamente da bubble (diagramma delle bubble) che verranno fornite nell'editor come se fossero delle attività specifiche.

3.2 Architettura server

Il server (parte back-end$_G$) è sviluppato in Node.js ed offre i seguenti servizi:

- Fornire la Single Page Application ai client che la richiedono;
- Fornire la lista di bubble utilizzabili nell'editor;
- Generare il codice sorgente, nel formato desiderato dal client, del progetto inviatogli.

In particolare, la componente che genera il codice sorgente è stata realizzata utilizzando un'architettura di tipo Pipe And Filter, in modo tale da assegnare un compito ben preciso ad ogni modulo per attuare una procedura sequenziale a "catena di montaggio". L'ultimo modulo ha il compito di creare un file compresso .zip del codice generato che sarà poi inviato al client.
Le bubble saranno salvate in una base dati per poter garantire una futura estendibilità del numero di queste ultime, eventualmente anche in altri domini da quello considerato al momento (i giochi da tavolo).

3.2.1 Comunicazioni server-client

La Single Page Application viene fornita al client semplicemente attraverso una pagina HTML.
Per la richiesta e fornitura delle bubble, client e server utilizzano AJAX per lo scambio di dati in formato JSON in modo tale da alleggerire il traffico.
Per la richiesta della generazione del codice, il client invia i dati del progetto in formato JSON utilizzando AJAX ed il server una volta elaborata la richiesta procede con l'inviare il file .zip precedentemente descritto.

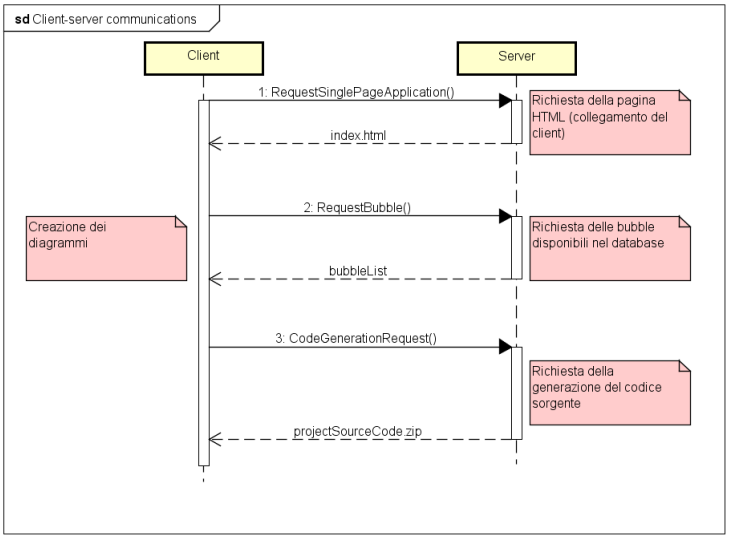

Figura 1: Esempi delle possibili comunicazioni client-server

4 Componenti e classi principali

4.1 SweDesigner

I package contenuti al suo interno sono:

- SweDesigner::Client;
- SweDesigner::Server;

Questo package non contiene delle classi.

4.2 SweDesigner::Client

I package contenuti al suo interno sono:

- SweDesigner::Client::Model;
- SweDesigner::Client::View;

Questo package non contiene delle classi.

4.3 SweDesigner::Client::Model

I package contenuti al suo interno sono:

- SweDesigner::Client::Model::RequestHandler;

Le classi contenute al suo interno verranno elencate qui di seguito.

4.3.1 SweDesigner::Client::Model::Command

È l'interfaccia che rappresenta un generico comando impartito dai moduli View ai Model.
FAN-IN:

- ConcreteCommand;
- MainView;
- State;

Non ci sono dipendenze OUT.

4.3.2 SweDesigner::Client::Model::ConcreteCommand

Implementa l'interfaccia Command per la rappresentazione concreta dei singoli comandi impartiti dai moduli View ai Model. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.3.3 SweDesigner::Client::Model::State

Gestisce la cronologia delle operazioni svolte permettendo le operazioni di unDo e reDo. FAN-IN:

- MainView;

Non ci sono dipendenze OUT.

4.3.4 SweDesigner::Client::Model::DAO

Si occupa della persistenza dei dati, in particolare del salvataggio su file system locale del progetto già esistente. FAN-IN:

- MainView;

Non ci sono dipendenze OUT.

4.3.5 SweDesigner::Client::Model::MainModel

È il componente del programma che si occupa di gestire la parte logica dell'editor. FAN-IN:

- ConcreteCommand;
- DAO;
- MainView;

Non ci sono dipendenze OUT.

4.3.6 SweDesigner::Client::Model::TitleBarModel

È il componente del programma che si occupa di gestire la parte logica della barra del titolo. FAN-IN:

- MainModel;

Non ci sono dipendenze OUT.

4.3.7 SweDesigner::Client::Model::ToolBarModel

È il componente del programma che si occupa di gestire la parte logica delle toolbar. FAN-IN:

- MainModel;

Non ci sono dipendenze OUT.

4.3.8 SweDesigner::Client::Model::PackageToolbar

Rappresenta la particolare toolbar legata all'editor del diagramma dei package. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.3.9 SweDesigner::Client::Model::ClassToolbar

Rappresenta la particolare toolbar legata all'editor del diagramma delle classi. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.3.10 SweDesigner::Client::Model::ActivityToolbar

Rappresenta la particolare toolbar legata all'editor del diagramma delle attività. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.3.11 SweDesigner::Client::Model::BubbleToolbar

Rappresenta la particolare toolbar legata all'editor del bubble flowchart. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.3.12 SweDesigner::Client::Model::AddressModel

È il componente del programma che si occupa di gestire la parte logica della barra degli indirizzi. FAN-IN:

- MainModel;

Non ci sono dipendenze OUT.

4.3.13 SweDesigner::Client::Model::EditPanelModel

È il componente del programma che si occupa di gestire la parte logica del pannello di editing later FAN-IN:

- ItemPanel;
- MainModel;

Non ci sono dipendenze OUT.

4.3.14 SweDesigner::Client::Model::ItemPanel

Estende la funzionalità di EditPanelModel specificamente per ciascun oggetto selezionato. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.3.15 SweDesigner::Client::Model::DiagramTree

È la collezione di tutti i model associati a ciascun diagramma. FAN-IN:

- MainModel;

Non ci sono dipendenze OUT.

4.3.16 SweDesigner::Client::Model::Diagram

Si occupa di gestire la parte logica di un diagramma. FAN-IN:

- ActivityDiagram;
- BubbleDiagram;
- ClassDiagram;
- DiagramTree;
- PackageDiagram;

Non ci sono dipendenze OUT.

4.3.17 SweDesigner::Client::Model::PackageDiagram

Estende le funzionalità di Diagram per rappresentare un diagramma dei package. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.3.18 SweDesigner::Client::Model::ClassDiagram

Estende le funzionalità di Diagram per rappresentare un diagramma delle classi. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.3.19 SweDesigner::Client::Model::ActivityDiagram

Estende le funzionalità di Diagram per rappresentare un diagramma delle attività. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.3.20 SweDesigner::Client::Model::BubbleDiagram

Estende le funzionalità di Diagram per rappresentare un bubble flowchart. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.4 SweDesigner::Client::Model::RequestHandler

Questo package non contiene dei sottopackage. Le classi contenute al suo interno verranno elencate qui di seguito.

4.4.1 SweDesigner::Client::Model::RequestHandler::Sender

Si occupa di gestire le comunicazioni in uscita verso il server. FAN-IN:

- MainModel;

Non ci sono dipendenze OUT.

4.4.2 SweDesigner::Client::Model::RequestHandler::Receiver

Si occupa di gestire le comunicazioni in entrata dal server. FAN-IN:

- Sender;

Non ci sono dipendenze OUT.

4.5 SweDesigner::Client::View

Questo package non contiene dei sottopackage. Le classi contenute al suo interno verranno elencate qui di seguito.

4.5.1 SweDesigner::Client::View::MainView

È il componente del programma che si occupa di gestire l'interfaccia grafica. Nella particolare declinazione MVC adottata da Backbone.js, si occupa anche di gestire gli input dell'utente e si interfaccia con il model attraverso dei command. È un aggregatore di altre classi View specializzate nella gestione dei diversi elementi dell'interfaccia grafica, in particolare TitleBarView, ToolBarView, AddressView, EditPanelView e Paper. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.5.2 SweDesigner::Client::View::TitleBarView

È il componente del programma che fa la funzione di view per la barra del titolo, dove saranno collocati il menu dell'applicazione e gli shortcut. FAN-IN:

- MainView;

Non ci sono dipendenze OUT.

4.5.3 SweDesigner::Client::View::ToolBarView

È il componente del programma che fa la funzione di view per la toolbar dove saranno collocati gli strumenti per editare i diagrammi. FAN-IN:

- MainView;

Non ci sono dipendenze OUT.

4.5.4 SweDesigner::Client::View::AddressView

È il componente del programma che fa la funzione di view per il cosiddetto breadcrumb dove viene inserita la posizione corrente. FAN-IN:

- MainView;

Non ci sono dipendenze OUT.

4.5.5 SweDesigner::Client::View::EditPanelView

È il componente del programma che fa la funzione di view per le informazioni editabili degli elementi che fanno parte dei diversi diagrammi. FAN-IN:

- MainView;

Non ci sono dipendenze OUT.

4.5.6 SweDesigner::Client::View::Paper

È il componente del programma che fa la funzione di view per i diversi diagrammi. FAN-IN:

- MainView;

Non ci sono dipendenze OUT.

4.6 SweDesigner::Server

I package contenuti al suo interno sono:

- SweDesigner::Server::CodeGenerator;
- SweDesigner::Server::DAORequestHandler;
- SweDesigner::Server::RequestHandler;

Questo package non contiene delle classi.

4.7 SweDesigner::Server::CodeGenerator

I package contenuti al suo interno sono:

- SweDesigner::Server::CodeGenerator::Builder;
- SweDesigner::Server::CodeGenerator::Coder;
- SweDesigner::Server::CodeGenerator::Parser;
- SweDesigner::Server::CodeGenerator::Zipper;

Le classi contenute al suo interno verranno elencate qui di seguito.

4.7.1 SweDesigner::Server::CodeGenerator::CodeGenerator

E' il componente che rende disponibile la funzionalità per cui, dato un file valido in formato JSON, restituisce un pacchetto in formato .zip contenente i file del codice sorgente che costituiscono il programma rappresentato dal file in input; i file prodotti sono strutturati in packages, come indicato nel file JSON in input. FAN-IN:

- Sender;

Non ci sono dipendenze OUT.

4.8 SweDesigner::Server::CodeGenerator::Builder

Questo package non contiene dei sottopackage. Le classi contenute al suo interno verranno elencate qui di seguito.

4.8.1 SweDesigner::Server::CodeGenerator::Builder::Builder

È il componente che rende disponibile la funzionalità, dato un file JSON in input che rappresenti un programma, di ottenere un oggetto contenitore del codice sorgente corrispondente al contenuto del file di input. Tale codice è suddiviso e strutturato come indicato nel file di input. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.9 SweDesigner::Server::CodeGenerator::Coder

Questo package non contiene dei sottopackage. Le classi contenute al suo interno verranno elencate qui di seguito.

4.9.1 SweDesigner::Server::CodeGenerator::Coder::JavaCoder

È il componente che rende disponibile la funzionalità, dato un oggetto in input che rappresenta un file JSON parsificato, di ottenere un oggetto contenente il codice sorgente, in linguaggio Java, corrispondente all'oggetto in input. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.9.2 SweDesigner::Server::CodeGenerator::Coder::JavascriptCoder

È il componente che rende disponibile la funzionalità, dato un oggetto in input che rappresenta un file JSON parsificato, di ottenere un oggetto contenente il codice sorgente, in linguaggio Javascript, corrispondente all'oggetto in input. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.9.3 SweDesigner::Server::CodeGenerator::Coder::CoderClass

È il componente che mette a disposizione la funzionalità, data una stringa in input in formato JSON che rappresenta una classe valida, di ottenere il corrispondente codice sorgente di tale classe. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.9.4 SweDesigner::Server::CodeGenerator::Coder::CoderOperation

È il componente che mette a disposizione la funzionalità, data una stringa in input in formato JSON che rappresenta un'operazione valida, di ottenere il corrispondente codice sorgente di tale operazione. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.9.5 SweDesigner::Server::CodeGenerator::Coder::CodeParameter

È il componente che mette a disposizione la funzionalità, data una stringa in input in formato JSON che rappresenta un parametro di una lista valido, di ottenere il corrispondente codice sorgente di tale parametro. È possibile scegliere fra la codifica in Java o Javascript. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.9.6 SweDesigner::Server::CodeGenerator::Coder::CoderActivity

È il componente che mette a disposizione la funzionalità, data una stringa in input in formato JSON che rappresenta un diagramma delle attività valido, di ottenere il corrispondente codice sorgente di tale attività. È possibile scegliere fra la codifica in Java o Javascript. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.9.7 SweDesigner::Server::CodeGenerator::Coder::CodedProgram

È il componente che contiene il codice sorgente prodotto dal Coder. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.9.8 SweDesigner::Server::CodeGenerator::Coder::Coder

Componente che funge da interfaccia alle operazioni di codifica di una stringa, in formato JSON che rappresenta un programma valido; tali operazioni permettono di ottenere un oggetto contenente il codice sorgente, in Java o Javascript, corrispondente alla stringa in input. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.9.9 SweDesigner::Server::CodeGenerator::Coder::CoderElement

Componente astratta che offre la funzionalità di ottenere, data una stringa in input in formato JSON che rappresenta un elemento di classe valido, il corrispondente codice sorgente, in Java o Javascript. FAN-IN:

- Coder;

Non ci sono dipendenze OUT.

4.10 SweDesigner::Server::CodeGenerator::Parser

Questo package non contiene dei sottopackage. Le classi contenute al suo interno verranno elencate qui di seguito.

4.10.1 SweDesigner::Server::CodeGenerator::Parser::Parser

È il componente che rende disponibile la funzionalità, dato un file JSON valido in input, di ottenere un oggetto contenente le informazioni che costituiscono il file in input. FAN-IN:

- CodeGenerator;

Non ci sono dipendenze OUT.

4.11 SweDesigner::Server::CodeGenerator::Zipper

Questo package non contiene dei sottopackage. Le classi contenute al suo interno verranno elencate qui di seguito.

4.11.1 SweDesigner::Server::CodeGenerator::Zipper::Zipper

E' il componente che rende disponibile la funzionalità per cui, dato un file valido in formato JSON, restituisce un pacchetto in formato .zip contenente i file del codice sorgente che costituiscono il programma rappresentato dal file in input; i file prodotti sono strutturati in packages, come indicato nel file JSON in input. Non ci sono dipendenze IN.Non ci sono dipendenze OUT.

4.12 SweDesigner::Server::DAORequestHandler

Questo package non contiene dei sottopackage. Questo package non contiene delle classi.

4.13 SweDesigner::Server::RequestHandler

Questo package non contiene dei sottopackage. Le classi contenute al suo interno verranno elencate qui di seguito.

4.13.1 SweDesigner::Server::RequestHandler::Sender

Si occupa di gestire le comunicazioni in uscita verso il client. FAN-IN:

- Zipper;

Non ci sono dipendenze OUT.

4.13.2 SweDesigner::Server::RequestHandler::Receiver

Si occupa di gestire le comunicazioni in entrata dal client. FAN-IN:

- Sender;

Non ci sono dipendenze OUT.

4.14 Tracciamento Classi-Requisiti

Classe	Requisiti
Classeschiiiiiiiiiiiiiiiiiiiiiiiiiifo	RSCHF

Tabella 2

A Design pattern utilizzati

A.1 Strutturali

A.1.1 Facade

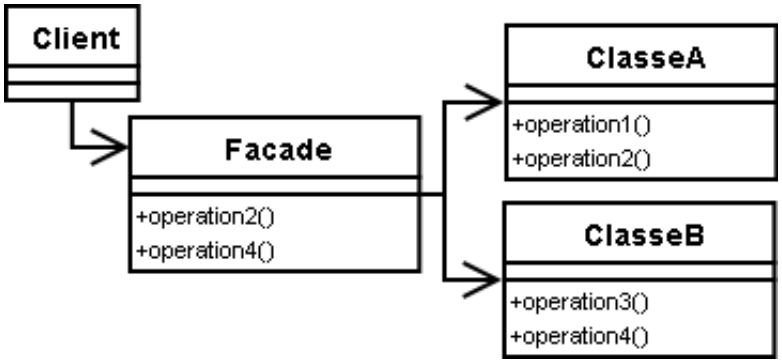

Figura 2: Esempio pattern Facade

Scopo
Fornire un'interfaccia per l'accesso ad uno o più sottosistemi complessi nascondendone la complessità all'esterno.

Problema
Una parte del client necessita di una interfaccia semplificata alla funzionalità di un sottosistema complesso.

Struttura
La principale componente individuabile è la classe Facade che si interpone tra il sottosistema e l'esterno, ed associa ogni richiesta ad una classe del sottosistema, delegando la risposta.

Utilizzo nel progetto
Utilizzato, ad esempio, lato server nel CodeGenerator per interfacciarsi con le diverse componenti.

A.2 Creazionali

A.2.1 Singleton

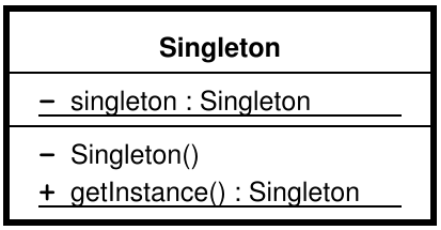

Figura 3: Esempio pattern Singleton

Scopo

Garantire che venga creata una ed una sola istanza di una determinata classe e di fornirne un punto di accesso globale.

Problema

Il sistema richiede una ed una istanza di un oggetto; inoltre, sono necessari un accesso globale e la possibilità di effettuare un'inizializzazione pigra dell'istanza.

Struttura

L'unica componente è la classe Singleton che definisce i metodi opportuni per permettere la creazione di un'unica istanza e l'accesso a quest'ultima.

Utilizzo nel progetto

Utilizzato, ad esempio, lato client per fornire un'unica esistenza della classe State interna al Model.

A.2.2 Factory

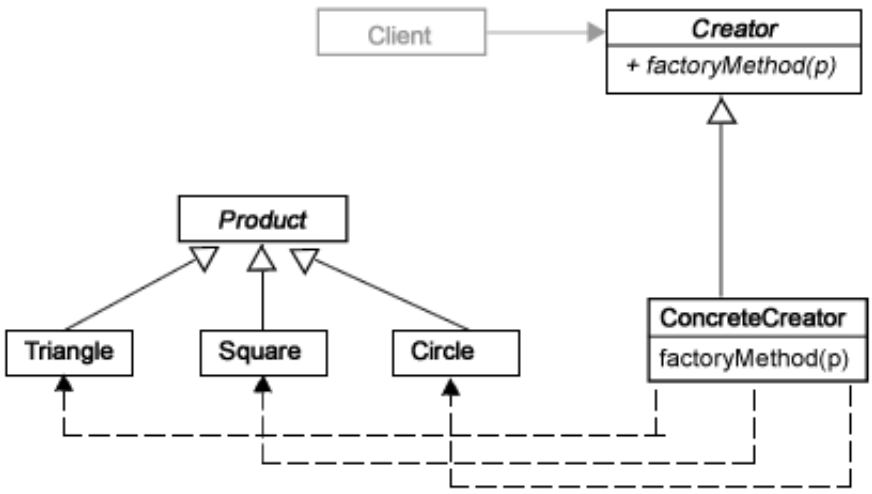

Figura 4: Esempio pattern Factory

Scopo
Permettere la creazione di oggetti senza esporre la logica creazionale al client mediante l'utilizzo di un'interfaccia.

Problema
Un'applicazione per essere portabile ha bisogno di incapsulare le dipendenze delle componenti del sistema in modo efficace ed efficiente.

Struttura
Sono individuabili tre componenti principali:

- Product: interfaccia di creazione dei prodotti;
- ConcreteProduct: implementa l'interfaccia secondo i metodi definiti;
- ConcreteCreator: definisce il factory method che ritornerà l'oggetto appropriato.

Utilizzo nel progetto
Utilizzato, ad esempio, lato server nella componente Coder per separare la costruzione degli oggetti generati.

A.3 Comportamentali

A.3.1 Observer

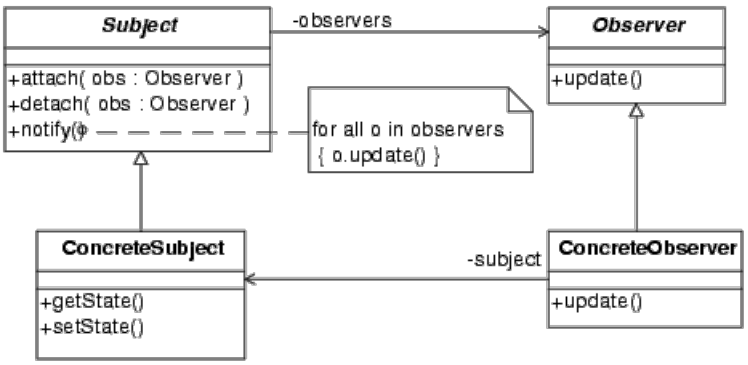

Figura 5: Esempio pattern Observer

Scopo
Definire una dipendenza "1..n" fra oggetti, riflettendo le modifiche dell'oggetto sui suoi dipendenti.

Problema
È necessario implementare un sistema di gestione di eventi provenienti da diversi oggetti.

Struttura
Sono individuabili quattro componenti principali:

- Subject: interfaccia per permettere agli osservatori di sottoscriversi e cancellarsi, avendo un riferimento ad ognuno di quelli iscritti;
- ConcreteSubject: mantiene lo stato di un oggetto concreto, notificando gli osservatori concreti in caso di cambiamenti;
- Observer: interfaccia per consentire agli osservatori di aggiornarsi al cambiamento di stato dell'oggetto osservato;
- ConcreteObserver: implementa l'interfaccia definita dall'Observer esplicitando le azioni da eseguire qualora si verifichi un cambio di stato dell'oggetto osservato.

Utilizzo nel progetto
Utilizzato, ad esempio, lato client dalla View per osservare i cambiamenti del Model; la sua implementazione è fornita da Backbone.js.

A.3.2 Command

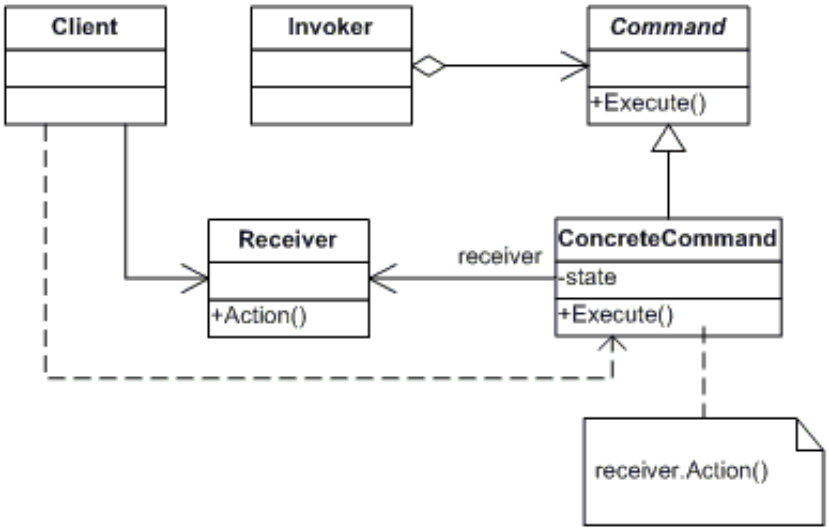

Figura 6: Esempio pattern Command

Scopo
Incapsulare il codice che effettua un'azione separandolo dall'oggetto che ne richiede l'esecuzione.

Problema
È necessario inviare richieste agli oggetti senza conoscere l'operazione richiesta o il destinatario della richiesta.

Struttura
Sono individuabili cinque componenti principali:

- Client: effettua la richiesta del comando e imposta il Receiver;
- Receiver: conosce come effettuare il comando;
- Invoker: effettua l'invocazione del comando;
- Command: interfaccia dei comandi;
- ConcreteCommand: implementa il comando e invoca l'operazione sul Receiver.

Utilizzo nel progetto
Utilizzato, ad esempio, lato client dove ogni intervento della View sul Model sarà trasmesso a quest'ultimo attraverso un command avente metodi di "exec()" e "undo()".