Sirius

Sequenziatore

Specifica Tecnica

Versione 1.0.0

Ingegneria Del Software AA 2013-2014

Informazioni documento

Titolo documento:	Specifica Tecnica
Data creazione:	2014-02-12
Versione attuale:	1.0.0
Utilizzo:	Esterno
Nome file:	*SpecificaTecnica_v1.0.0.pdf*
Redazione:	Quaglio Davide
Approvazione:	Giachin Vanni
Distribuito da:	Sirius
Destinato a:	Prof. Vardanega Tullio
	Prof. Cardin Riccardo
	Zucchetti S.p.A

Sommario

Descrizione dell'architettura e dei componenti relativi allo sviluppo del progetto *Sequenziatore*.

Diario delle modifiche

Versione	Data	Autore	Ruolo	Descrizione
0.0.1	*2014-03-15*	*Giachin Vanni*	*?*	Stesura introduzione

Indice

1 Introduzione

1.1 Scopo del Documento

Lo scopo di questo documento è la definizione delle specifiche progettuali del prodotto *software Sequenziatore*.
Viene quindi presentata l'architettura ad alto livello del sistema, e la descrizione delle singole componenti e dei *design pattern*$_G$ utilizzati.

1.2 Scopo del Prodotto

Lo scopo del progetto *Sequenziatore*, è di fornire un servizio di gestione di processi definiti da una serie di passi da eseguirsi in sequenza o senza un ordine predefinito, utilizzabile da dispositivi mobili di tipo *smaptphone* o *tablet*.

1.3 Glossario

Al fine di rendere più leggibili e comprensibili i documenti, i termini tecnici, di dominio, gli acronimi e le parole che necessitano di essere chiarite, sono riportate nel documento *Glossario_v1.0.0.pdf*.
Ciascuna occorrenza dei vocaboli presenti nel *Glossario* è seguita da una "G" maiuscola in pedice.

1.4 Riferimenti

1.4.1 Normativi

- Norme di Progetto: *NormeDiProgetto_v1.0.0.pdf*.
- Analisi dei Requisiti: *AnalisiDeiRequisiti_v1.0.0.pdf*.

1.4.2 Informativi

- Design Patterns: Elementi per il riuso di software ad oggetti - Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (2002);
- Learning JavaScript Design Patterns, Addy Osmani, Volume 1.5.2: http://addyosmani.com/resources/essentialjsdesignpatterns/book;
- Regolamento dei documenti, prof. Vardanega Tullio: http://www.math.unipd.it/~tullio/IS-1/2013/;
- Dispense di ingegneria del software modulo A:
  - Progettazione software, prof. Vardanega Tullio: http://www.math.unipd.it/~tullio/IS-1/2013/Dispense/P09.pdf;

  - Diagrammi delle classi e degli oggetti, prof. Cardin Riccardo: http://www.math.unipd.it/~tullio/IS-1/2013/Dispense/E02a.pdf;
  - Diagrammi di sequenza, prof. Cardin Riccardo: http://www.math.unipd.it/~tullio/IS-1/2013/Dispense/E03a.pdf;
  - Diagrammi di attività, prof. Cardin Riccardo: http://www.math.unipd.it/~tullio/IS-1/2013/Dispense/E03b.pdf;
  - Introduzione ai design pattern, prof. Cardin Riccardo: http://www.math.unipd.it/~tullio/IS-1/2013/Dispense/E04.pdf;
  - Diagrammi dei package, prof. Cardin Riccardo: http://www.math.unipd.it/~tullio/IS-1/2013/Dispense/E05.pdf;

- Dispense di ingegneria del software modulo B:

  - Design pattern: Model-View-Controller, prof. Cardin Riccardo: http://www.math.unipd.it/~rcardin/pdf/Design%20Pattern%20-%20Model%20View%20Controller_4x4.pdf;
  - Design pattern strutturali, prof. Cardin Riccardo: http://www.math.unipd.it/~rcardin/pdf/Design%20Pattern%20Strutturali_4x4.pdf;
  - Design pattern creazionali, prof. Cardin Riccardo: http://www.math.unipd.it/~rcardin/pdf/Design%20Pattern%20Creazionali_4x4.pdf;
  - Design pattern comportamentali, prof. Cardin Riccardo: http://www.math.unipd.it/~rcardin/pdf/Design%20Pattern%20Comportamentali_4x4.pdf;
  - Esercizi sugli errori rilevati in RP, prof. Cardin Riccardo: http://www.math.unipd.it/~rcardin/pdf/Esercitazione%20-%20Errori%20comuni%20RP_4x4.pdf;

2 Definizione dell' architettura

2.1 Metodo e formalismo di specifica

L' architettura del sistema è la struttura del sistema, che comprende gli elementi *software*, la visibilità esterna di questi elementi e la relazione tra loro. Questo documento andrà ad esporre le componenti di alto livello del sistema che verranno poi approfondite nel periodo di Progettazione di dettaglio e codifica, per analizzare l' architettura del sistema il *Sequenziatore*si seguirà l' approccio *top-down*, quindi innanzitutto si analizzerà il sistema fornendone una descrizione generale per poi scomporre le varie parti andando sempre più in dettaglio analizzando le singole componenti. Successivamente si analizzeranno i *design pattern* adottati e come verranno implementati. Per esporre al meglio l architettura del sistema e il suo funzionamento di alto livello si utilizzeranno diagrammi dei *package*, delle classi, di attività e di sequenza seguendo quanto imposto dalle *NormeDiProgetto_v1.0.0.pdf* .

3 Design pattern

3.1 Design pattern architetturali

3.1.1 Model View Presenter

- **Scopo:** Il *pattern*$_G$ architetturale *Model View Presenter* (MVP) è un derivato del *Model View Controller* (MVC), focalizzato sulla valorizzazione della logica della presentazione. Entrambi i pattern hanno lo sopo di disaccoppiare la logica dell'applicazione dalla rappresentazione grafica.
  Il *pattern*$_G$ MVP prevede la suddivisione dell'applicazione in tre componenti:
  - **Model:** Definisce il modello dati e le regole di accesso e di modifica;
  - **View:** Si occupa della rappresentazione dell'interfaccia utente;
  - **Presenter:** Contiene la logica dell'applicazione, si occupa delle comunicazioni tra vista e modello e dell'aggiornamento della vista.
- **Contesto d'uso:**

3.2 Design pattern strutturali

3.2.1 Adapter

- **Scopo:** Il *pattern*$_G$ strutturale *Adapter* permette di utilizzare un componente software la cui interfaccia deve essere adattata per potersi integrare ad un'altra presente nell'applicazione esistente.
  Tale *pattern*$_G$ può essere basato sia su classi che su oggetti, perciò, l'istanza della classe da adattare, può derivare tramite ereditarietà o composizione.
- **Contesto d'uso:**

3.2.2 Decorator

- **Scopo:** Il *pattern*$_G$ strutturale *Decorator* permette di aggiungere dinamicamente funzionalità ad un oggetto base, con la possibilità di comporle arbitrariamente.
  Tale *pattern*$_G$ si pone come alternativa all'uso dell'ereditarietà singola o multipla;
- **Contesto d'uso:**

3.2.3 Facade

- **Scopo:** Il *pattern*$_G$ strutturale *Facade* prevede l'utilizzo di un'interfaccia unica e semplice per un sottosistema complesso, diminuendo la complessità del sistema;
- **Contesto d'uso:**

3.2.4 Proxy

- **Scopo:** Il *pattern*$_G$ strutturale *Proxy* viene utilizzato per accedere ad un un oggetto complesso di cui si vogliono controllare gli accessi, tramite un oggetto semplice, che espone gli stessi metodi dell'oggetto che maschera;
- **Contesto d'uso:**

3.3 Design pattern creazionali

3.3.1 Singleton

- **Scopo:** Il *pattern*$_G$ creazionale *Singleton* viene utilizzato quando si ha la necessità di avere una sola istanza di una classe e di avere un punto di accesso globale ad essa;
- **Contesto d'uso:**

3.3.2 Abstract Factory

- **Scopo:** Il *pattern*$_G$ creazionale *Abstract Factory* fornisce un'interfaccia per creare famiglie di prodotti senza specificare classi concrete. Le classi che concretizzano tale interfaccia, vengono costruite una sola volta, e consentono di utilizzare una varietà di elementi che presentano le stesse funzionalità con diverse implementazioni;
- **Contesto d'uso:**

3.4 Design pattern comportamentali

3.4.1 Command

- **Scopo:** Il *pattern*$_G$ comportamentale *Command* permette di separare l'invocazione di un comando dai suoi dettagli implementativi;
- **Contesto d'uso:**

3.4.2 Iterator

- **Scopo:** Il *pattern*$_G$ comportamentale *Iterator* fornisce l'accesso sequenziale agli elementi di un aggregato senza esporne l'implementazione;
- **Contesto d'uso:**

3.4.3 Observer

- **Scopo:** Il *pattern*$_G$ comportamentale *Observer* viene utilizzato quando si vuole realizzare una dipendenza tra un soggetto e più oggetti, in cui il cambiamento di stato del un soggetto, viene notificato a tutti gli oggetti dipendenti;
- **Contesto d'uso:**

3.4.4 Strategy

- **Scopo:** Il *pattern*$_G$ comportamentale *Strategy* viene utilizzato per definire una famiglia di algoritmi, incapsularli e renderli intercambiabili;
- **Contesto d'uso:**

3.4.5 Template method

- **Scopo:** Il *pattern*$_G$ comportamentale *Template method* viene utilizzato per definire lo scheletro di un algoritmo, lasciando l'implementazione di alcuni passi alle sottoclassi. In particolare consente di specificare l'ordine delle operazioni da effettuare ma di delegare la loro implementazione o parte di essa alle sottoclassi;
- **Contesto d'uso:**