Progetto SIGEOL

Definizione di prodotto

v0.1.0

Redazione:

2 marzo 2009

quixoft.sol@gmail.com

Verifica:	Verificatore
Approvazione:	Responsabile
Stato:	Preliminare — Formale
Uso:	Interno — Esterno
Distribuzione:	QuiXoft

Sommario

Descrizione dettagliata delle caratteristiche tecniche ed architetturali del prodotto Sigeol

Indice

1 Introduzione

1.1 Scopo del documento

Il presente documento denominato Descrizione di prodotto si prefigge di illustrare ed analizzare con maggior dettaglio i metodi ed i formalismi adottati nella definizione del prodotto Sigeol

1.2 Scopo del prodotto

Il progetto sotto analisi, denominato Sigeol, si prefigge di automatizzare la generazione, la gestione, l'ottimizzazione e la consultazione degli orari di lezione. Per maggiori dettagli consultare il documento denominato Analisi dei Requisiti alla sua ultima versione.

1.3 Glossario

Le definizioni dei termini specialistici usati nella stesura di questo e di tutti gli altri documenti possono essere trovate nel documento denominato Glossario al fine di eliminare ogni ambiguità e di facilitare la comprensione dei temi trattati. Ogni termine la cui definizione è disponibile all'interno del glossario verrà marcato con una sottolineatura.

1.4 Riferimenti normativi

Il documento denominato Norme di Progetto accompagna e complementa il presente ed ogni documento ufficiale.

2 Standard di progetto

2.1 Standard di progettazione architetturale

La definizione dell'intero sistema oggetto di studio è stata effetuata attraverso l'uso di diagrammi UML e l'applicazione di pattern consolidati ed in uso in molti prodotti software.

2.1.1 UML

Il linguaggio UML è utilizzato per la modellazione architetturale di un sistema in quanto grazie alla sua capacità e chiarezza espressiva risulta di facile comprensione anche a persone esterne al progetto stesso. Il team QuiXoft ha utilizzato UML 2.0 per:

- Diagrammi use-case nel documento Analisi dei Requisiti
- Diagrammi delle classi, delle componenti, di attività e delle sequenze nei documenti Specifica tecnica e Definizione di prodotto

2.2 Pattern

All'interno dell'architettura del sistema sono stati utilizzati i seguenti pattern, presenti nel framework Ruby on Rails il quale è alla base di tutto il prodotto:

MVC Il pattern MVC (Model View Controller) si basa sulla separazione tra i componenti software del sistema, che gestiscono il modo in cui presentare i dati, e i componenti che gestiscono i dati stessi.

Façade Permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi aventi interfacce complesse e molto diverse tra loro, nonché a blocchi di codice complessi.

REST Representational state transfer (REST) è un tipo di architettura software per i sistemi di ipertesto distribuiti come il World Wide Web. REST si riferisce ad un insieme di principi di architetture di rete, i quali delineano come le risorse sono definite e indirizzate.

Convention Over Configuration Convention Over Configuration è un paradigma di programmazione che prevede configurazione minima (o addirittura assente) per il programmatore che utilizza un framework che lo rispetti, obbligandolo a configurare solo gli aspetti che si differenziano dalle implementazioni standard o che non rispettano particolari convenzioni di denominazione o simili. Significa che Rails prevede delle impostazioni di default per qualsiasi aspetto dell'applicazione. Utilizzando queste convenzioni sarà possibile velocizzare i tempi di sviluppo evitando di realizzare scomodi file di configurazione. L'esempio più chiaro del COC si può notare a livello di modelli: rispettando le convenzioni previste dal framework è possibile realizzare strutture di dati complesse con molte relazioni tra oggetti in pochissimo tempo, in maniera quasi meccanica e soprattutto senza definire nessuna configurazione. Questo concetto differenzia Rails dai framework che prevedono molte righe di configurazione per ogni aspetto dell'applicazione. Con il COC tutto diventa più snello e più dinamico. Ovviamente per situazioni in cui le convenzioni non possano essere rispettate, Rails permette di utilizzare schemi funzionali diversi da quelli previsti.

DRY Questo concetto, fortemente filosofico, prevede che ciascun elemento di un'applicazione debba essere implementato solamente una volta e niente debba essere ripetuto. Questo significa che, mediante Rails, è possibile gestire funzionalità ripetitive con una estrema fattorizzazione del codice ("scrivo una volta e uso più volte") che facilita sia lo sviluppo iniziale che eventuali modifiche successive del prodotto.

View Helper Questo pattern disaccoppia il Business Logic dallo strato

View, il che facilita la manutenibilità. Aiuta a separare, in fase di sviluppo, la responsabilità del web designer e dello sviluppatore.

Active Record Secondo il pattern Active Record esiste una relazione molto stretta fra tabella e classe, colonne e attributi della classe.

- una tabella di un database relazionale è gestita attraverso una classe
- una singola istanza della classe corrisponde ad una riga della tabella
- alla creazione di una nuova istanza viene creata una nuova riga all'interno della tabella, e modificando l'istanza la riga viene aggiornata

2.3 Standard di documentazione del codice

Il team QuiXoft si avvalerà dello strumento RDoc, specifico per il linguaggio Ruby. Questo strumento estrapola dal codice sorgente i commenti al codice stesso, organizzandoli e rendendoli disponibili alla consultazione tramite pagine HTML. Per questo motivo ogni membro del team alla stesura di qualsiasi classe o metodo dovrà documentarlo tramite la sintassi specifica di RDoc.

2.4 Standard di denominazione di entità e relazioni

Lo schema di denominazione deve essere determinante per la comprensione del flusso logico dell'applicazione. Verrano quindi utilizzati nomi significativi che identifichino la funzione e lo scopo dell'elemento. Inoltre saranno seguite le convenzioni generali dello specifico linguaggio di programmazione utilizzato per realizzare l'elemento, nonchè ulteriori convenzioni dettate dal framework utilizzato. Per maggiori informazioni consultare la sezione 2.5

2.5 Standard di programmazione

Ogni file deve contenere esattamente una classe, eccezion fatta per i template di file HTML e JavaScript. Inoltre è necessaria ai fini di una migliore leggibilità, l'uso di una corretta indentazione, fornita dall'ambiente di sviluppo.

2.5.1 Ruby e framework Rails

Di seguito sono elencate le convenzioni utilizzate negli elementi sviluppati con il linguaggio Ruby.

Variabili locali

Prima lettera minuscola, seguita da altri caratteri minuscoli. Se la variabile comprende più parole, queste andranno separate con un _ (underscore). Esempio: `variabile_locale`

Variabili d'instanza

Si utilizza la stessa convenzione adottata nelle variabili locali, con l'aggiunta di un @ (at) prima del nome. Esempio: `@variabile_istanza`

Variabili di classe

Si utilizza la stessa convenzione adottata nelle variabili locali, con l'aggiunta di una doppia @ (at) prima del nome. Esempio: `@@variabile_classe`

Variabili globali

Si utilizza la stessa convenzione adottata nella variabili locali, con l'aggiunta di un `$` (dollar) prima del nome. Esempio: `$variabile_globale`

Costanti

Prima lettera maiuscola, seguita da altri caratteri maiuscoli. Se la variabile comprende più parole, queste andranno separate con un _ (underscore). Esempio: `UNA_COSTANTE`

Metodi d'istanza

Prima lettera minuscola, seguita da altri caratteri minuscoli. Se il nome comprende più parole, queste andranno separate con un _ (underscore). Esempio: `metodo_istanza`

Classi e moduli

Prima lettera maiuscola, seguita da altri caratteri minuscoli. Se il nome comprende più parole, la prima lettera di ogni parola deve essere maiuscola. Esempio: `UnaClasse`

Model

Si utilizza la stessa convenzione per le classi ed inoltre il nome dovrà essere il singolare (in lingua inglese) del nome della tabella del database a cui si riferisce. Esempio: `Order`

Controller

Si utilizza la stessa convenzione per le classi ed inoltre il nome dovrà essere il plurale (in lingua inglese) del nome del Model a cui si riferisce, seguito dalla parola *Controller*. Esempio: `OrdersController`

Tabelle del database

Prima lettera minuscola, seguita da altri caratteri minuscoli. Se il nome comprende più parola, queste andranno separate con un _ (underscore). Inoltre il nome deve essere il plurale del model (in lingua inglese) a cui la tabella si riferisce. Esempio: `orders`

Chiave primaria

Il nome della chiave primaria dovrà essere `id`

Chiavi esterne

Il nome della chiave esterna dovrà essere il singolare (in lingua inglese) della tabella di riferimento, seguito da un _ (underscore) e dalla parola *id*, con ogni carattere minuscolo. Esempio: `order_id`

Tabelle per le relazioni molti a molti

Concatenazione tramite _ (underscore) dei nomi al plurale (in lingua inglese) dei model coinvolti in ordine alfabetico, con ogni carattere minuscolo. Esempio `items_orders`

File

Ogni nome di file è caratterizzato dalla presenza di soli caratteri minuscoli e la concatenazione di più parole è effettuata tramite _ (underscore).

2.5.2 Java

Per quanto riguarda i file sorgenti scritti utilizzando il linguaggio Java fanno fede le norme e convenzioni acquisite da ogni membro del team durante il corso di Programmazione 3 o Programmazione concorrente e distribuita, a seconda dell'ordinamento a cui il componente appartiene.

2.6 Strumenti di lavoro

Durante tutto lo svolgimento del progetto, il team QuiXoft utilizzarà i seguenti strumenti:

- IDE NetBeans 6.5
  http://www.netbeans.org/
- JDK 6 Update 12
  http://java.sun.com/
- JRuby 1.1.5
  http://jruby.codehaus.org/
- GlassFishV3
  https://glassfish.dev.java.net/
- MySql 5.0
  http://www.mysql.com/
- Rails framework
  http://rubyonrails.org/
- RDoc
  http://rdoc.sourceforge.net/
- rcov
  http://rubyforge.org/projects/rcov/
- W3C validator
  http://validator.w3.org/
- Prototype 1.6
  http://www.prototypejs.org/
- LaTeX
  http://www.latex-project.org/

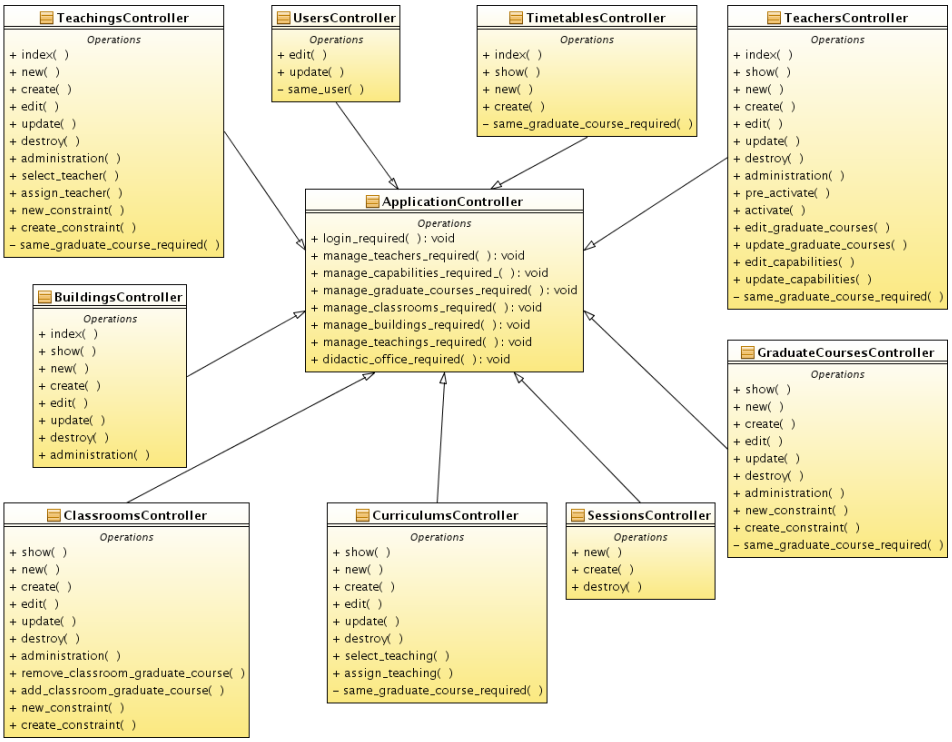
3 Specifica delle componenti

Il sistema Sigeol è strutturato seguendo il paradigma MVC, con l'aggiunta di ulteriori due componenti: MiddleMan e Algorithm.

3.1 Componente Controller

La componente Controller si occupa di gestire le azioni che l'utente effettua, solitamente attraverso una view. Ogni metodo pubblico rappresenta quindi una specifica azione, eccezion fatta per l'Application Controller. Di seguito è riportato il diagramma delle classi che illustra questa componente. Sono stati omessi volutamente i tipi di ritorno per i metodi che implementano un'azione, in quanto queste operazioni non sono utilizzate per

restituire un valore, bensì inizializzano alcune variabili d'istanza che saranno successivamente disponibili nella view specifica per quella azione. Inoltre per aumentare la leggibilità è stata omessa la derivazione della classe `ApplicationController` da `ActionController::Base`.



3.1.1 Azioni comuni a più controller

Per evitare fastidiose ripetizioni in questa sezione verranno descritti i metodi che figurano con lo stesso nome in diversi controller. La scelta dello stesso nome non è casuale, in quanto rispecchia la funzione dell'azione.

- `index`: rende disponibile alla view specifica un insieme d'istanze ed è raggiungibile eseguendo una richiesta GET all'indirizzo /nomecontroller. Ad esempio l'azione `index` di `GraduateCourseController` fornisce alla view un insieme di corsi di laurea ed è raggiungibile attraverso una richiesta GET all'indirizzo /graduate_courses.

- `show`: rende disponibile alla view specifica un'istanza ed è raggiungibile eseguendo una richiesta GET all'indirizzo /nomecontroller/id. Usando sempre l'esempio dei corsi di laurea, eseguendo una richiesta GET all'indirizzo /graduate_courses/1 verranno visualizzate le informazioni relative al corso di laurea con id 1.

- `new`: rende disponibile alla view specifica un'istanza vuota dello stesso tipo per permettere l'inserimento di un nuovo oggetto. E' raggiungibile eseguendo una richiesta GET /nomecontroller/new
- `create`: acquisisce i dati da una richiesta POST per salvare l'oggetto nel database attraverso il model. Solitamente i dati provengono da una form con metodo POST presente nella view per l'azione `new`. E' possibile comunque invocare questa azione mediante una richiesta POST all'indirizzo /nomecontroller
- `edit`: rende disponibile alla view specifica un'istanza esistente per permetterne la modifica. e' raggiungibile attraverso una richiesta GET all'indirizzo /nomecontroller/id/edit
- `update`: acquisisce i dati da una richiesta PUT per aggiornare lo stato dell'oggetto nel database attraverso il model. Solitamente i dati provengono da una form con metodo PUT presente nella view per l'azione `edit`. E' possibile comunque invocare questa azione mediante una richiesta PUT all'indirizzo /nomecontroller/id
- `destroy`: distrugge l'oggetto attraverso il model. Questa azione viene invocata tramite una richiesta DELETE all'indirizzo /nomecontroller/id
- `administration`: rende disponibile alla view specifica un insieme d'istanze per effettuarne l'amministrazione. Questa azione è raggiungibile attraverso una richiesta GET all'indirizzo /nomecontroller/administration.

Solitamente non è necessaria l'autenticazione o il possesso di alcuni privilegi per eseguire le azioni `index` e `show`. Per quanto riguarda gli altri metodi, invece, può ritenersi necessaria l'autenticazione od il possesso di alcuni privilegi. Per ogni controller sarà descritto questo aspetto. Differente invece è il caso del metodo `same_graduate_course_required`, dichiarato privato nei controllers che lo implementano. Questo metodo non rispecchia un'azione, ma è utilizzato come filtro, ovvero è chiamato prima o dopo una determinata azione, per impedire la modifica o la cancellazione di un oggetto appartenente ad un corso di laurea diverso da quello dell'utente autenticato.

3.1.2 ApplicationController

Questa classe deriva direttamente da `ActionController::Base`, ed è estesa da ogni controller. Prevede metodi di pubblica utilità per gli altri controller, ma nessuna azione. L' `ApplicationController` del sistema Sigeol prevede i seguenti metodi pubblici, utilizzati come filtri dagli altri controller.

- `login_required`: se l'utente non è autenticato questo metodo reindirizza alla pagina di login
- `manage_teachers_required`: se l'utente non possiede i privilegi per gestire i docenti questo metodo reindirizza alla pagina principale mostrando un errore.
- `manage_capabilities_required`: se l'utente non possiede i privilegi per gestire i privilegi questo metodo reindirizza alla pagina principale mostrando un errore.
- `manage_graduate_courses_required`: se l'utente non possiede i privilegi per gestire i corsi di laurea questo metodo reindirizza alla pagina principale mostrando un errore.
- `manage_classrooms_required`: se l'utente non possiede i privilegi per gestire le aule questo metodo reindirizza alla pagina principale mostrando un errore.
- `manage_buildings_required`: se l'utente non possiede i privilegi per gestire gli edifici questo metodo reindirizza alla pagina principale mostrando un errore.
- `manage_teachings_required`: se l'utente non possiede i privilegi per gestire gli insegnamenti questo metodo reindirizza alla pagina principale mostrando un errore.
- `didactic_office_required`: se l'utente non appartiene ad una segreteria didattica questo metodo reindirizza alla pagina principale mostrando un errore.

3.1.3 GraduateCoursesController

3.1.4 CurriculumsController

3.1.5 UsersController

3.1.6 TeachersController

3.1.7 SessionsController

3.1.8 TeachingsController

3.1.9 BuildingsController

3.1.10 ClassroomsController

3.1.11 TimetablesController

Diario delle modifiche

Data	Versione	Modifica
20 febbraio 2009	0.1.0	Stesura dell'indice