System Design

Introduzione

In questo documento analizzeremo l'architettura software, System Design e Object Design, e descrivermo i Design Pattern utilizzati, tutto rappresentato con diagrammi UML 2.0 (ClassDiagram, e SequenceDiagram).

Architettura Software

Per l'Architettura Software, abbiamo deciso di usare una architettura *Three-Tier,* utilizzando il Design Pattern **MVC**, per rappresentare i 3 livelli del software, questa scelta ci ha portato ad avere una flessibilità e modificabilità maggiore del sistema, poiché avendo più livelli, dove ogni livello svolge un compito ben preciso, e nessun altro, possiamo facilmente individuare i vari bug, e quindi modificare facilmente una componente senza che le altre vengano modificate di conseguenza, da qui deriva una potenziale riusabilità di componenti per sistemi diversi, i livello sono i seguenti:

- **Livello di presentazione:** Sono tutte le UI del sistema, le interazioni con l'utente, e come vengono rappresentate le informazioni.
- **Livello di Business:** In questo livello abbiamo tutte le regole di business, quindi il cuore dell'applicazione, controllerà i flussi di dato dal livello di presentazione e il data layer.
- **Livello dei Dati:** Qui invece troveremo tutte quelle classi che interagiranno con i dati veri e propri dell'applicazione, quindi con la nostra base di dati scelta.

Come si può notare il '*Livello di presentazione'* e il *'Livello dei Dati'* non sapranno la reale implementazione e neanche la loro ' esistenza ', per cui delle modifiche a uno dei due livelli non toccherà l'altro, e viceversa, mentre un discorso diverso per il *'Livello di Business'*, che dovendo regolare il flusso dei dati tra i due livello, verrà influenzato anche se di poco dai vari cambiamenti dei due livello.

Usando questo tipo di architettura, avremo un codice molto voluminoso, rinunciando anche in parte alla performance del sistema, preferendo una maggiore facilità di manutenzione e favorendo i futuri cambiamenti, relativamente ai cambiamenti, abbiamo avuto un'esperienza diretta, trovandoci a cambiare durante questa fase del progetto, a cambiare spesso implementazione e design , i cambiamento sono stati molto rapidi, ed hanno coinvolto poche classi del sistema.

Tecnologie Utilizzate:

- **JavaFX:** Software basato su Java, per Rich Internet Application, ci è tornato molto utile per la modellazione della parte grafica, utilizzando file FXML per la modellare la UI, e alla possibilità di integrare CSS al suo interno, e la possibilità di importare numerose librerie grafiche.
- **Java 8:** linguaggio di programmazione orientato agli oggetti utilizzato.
- **Oracle Database Express Edition 11g:** Questo è il nostro DBMS scelto per la memorizzazione dei dati nella nostra applicazione, lo abbiamo scelto per i vari servizi che offre Oracle e la grande compatibilità con Java.
- **SceneBuilder:** Programma di modellazione grafica per JavaFX, rendendo la modellazione UI molto intuitiva e semplice, lasciandoci più spazio per l'implementazione e la fase di testing del software.

- **GitHub**: E' stata la nostra scelta per il software di versioning, ci ha aiutato molto nel condividere e analizzare il codice degli altri componenti del gruppo. (https://github.com/JimMinor/EventManager2018)

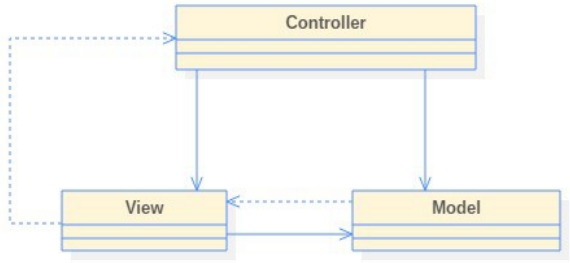
In Dettaglio, Design Pattern e Best Practices:

Abbiamo utilizzando in gran parte il **pattern architetturale MVC:**

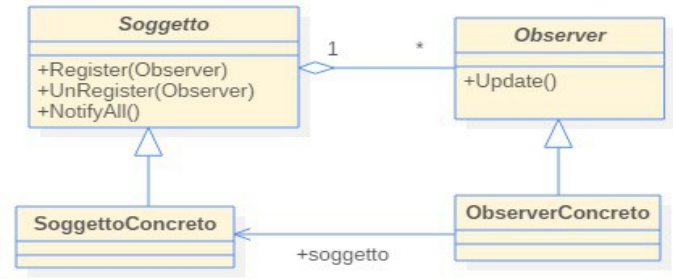
Per implentare parte della nostra architettura, sopratutto per quanto riguarda, i primi due livelli :

- **Presentazione**
- **Business**

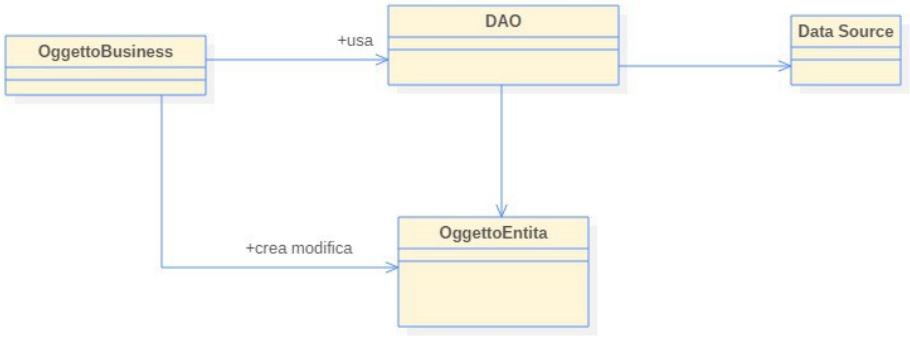
Questo ci ha portato a molti vantaggi, quali estendibilità, riusabilità, modalurità, e manutenibilità.



Come punto di partenza per implementare molte delle funzionalità del sistema, adattandolo alle tecnologie utilizzate.
Sopratutto l'implementazione della **View,** la quale è stata implementata con la consueta best practice, seguendo il Design Pattern **Obsever**.



Invece per il **Livello dei Dati**, abbiamo usato il Design Pattern **DAO**, il quale ci permetterà una maggiore facilità di manutenzione in caso di cambiamenti sulla basi di dati, e favorendo più implementazioni di basi di dati differenti, e diminuisce l'accoppiamento.


DAO
OggettoBusiness
+usa
Data Source
OggettoEntita
+crea modifica

Inoltre, abbiamo fatto uso della **Legge di Demetra**, per avere un *Basso Accoppiamento* tra le classi.

Dal Dominio all' Implementazione:

Una volta steso un documento del modello Funzionale e di Dominio, siamo entrati sempre nel più nel dettaglio del sistema, e implementando le funzionalità in base alle tecnologie scelte, per questo motivo abbiamo dovuto modificare alcuni Design Pattern.

MVC e JavaFX
JavaFX introduce i file FXML, e altre molte funzionalità, permettendo di avere una classe che controllerà il file FXML (che ne descriverà l'aspetto della UI), questa classe potrà essere sia come vista che come controller, ma questa scelta non ci è sembrata la più consona per il nostro obiettivo, per questo motivo abbiamo deciso di rimanere più vicino alla 'classica' implementazione del pattern **MVC**, abbiamo dunque utilizzato un file FXML, e abbiamo associato a quest'ultimo una classe con i soli riferimenti agli oggetti grafici, quali Button e TextField etc, impostando questa classe come controller della risorsa FXML, successivamente abbiamo creato un vero e proprio controller, il quale al suo interno avrà la **Business Logic**, e i Listener per ogni Component della View, in modo tale da avere tutta la logica in unica classe, che interagirà anche con gli oggetti di tipo **DAO** e **Entity**.
Dove necessario la **View** implementarà **Observer**, e il **Model** sarà l'oggetto **Observable**, e il **Controller** potrà essere visto come un **ActionListener**.

Data Layer e DAO:
Per implementare questo Layer, abbiamo si usato le classi DAO implementandole secondo il basi di dati utilizzata, ma queste classi sono state utilizzate solo per gestire gli errori (*SQLException)* lanciate dalle funzioni che comunicano direttamente con la base di dati.
Le query sono state gestite da classi apposite, perché abbiamo notato che alcune query potevano essere usate da più oggetti DAO che gestivano entità diverse, così da rendere le classi più leggibili e più facile di mantenere, mantenendo un basso accoppiamento.

Proseguiamo con la presentazione dei diagrammi UML:

Login:

Questa funzionalità è stata aggiunta in questa fase, un Impiegato o un Amministratore posso entrare nell'applicazione con le credenziali scelte nella parte Front-End della piattaforma, questo permetterà di 'chiudere' le aree riservate agli Impiegati.

E' stata usata un interfaccia *Autenticazione* in modo da facilitare futuri cambiamenti di autenticazione dell utente, è stata implementata un autenticazione con Username e Password.

CRC Card:

Nome Classe: *LoginStage*	**Package**: View
Superclasses	Initizializable,ControlledScreen
Subclasses	-
Responsabilità	**Collaboratori**
Questa classe si occupa di rappresentare la	*CambiaStage,Autenticazione,Impiegato*

schermata di login e, controllare la correttezza dei dati inseriti per il login, e di ritornare l'istanza 'Impiegato' corrispondente .	

Nome Classe: *CambiaStage*	**Package**: Control
Superclasses	-
Subclasses	-
Responsabilità	**Collaboratori**
Si occupa di cambiare gli Stage dell'applicazione	**-**

Nome Classe: *Autenticazione*	**Package**: DB
Superclasses	-
Subclasses	-
Responsabilità	**Collaboratori**
Controlla i dati per l'autenticazione	*ImpiegatoDAO,*

Nome Classe: *ImpiegatoDAOImp*	**Package**: DB
Superclasses	*ImpiegatoDAO*
Subclasses	
Responsabilità	**Collaboratori**
Operazioni di C.R.U.D sulla classe *Impiegato*	*GestoreQueryCerca*

Nome Classe: *GestoreQueryCerca*	**Package**: DB
Superclasses	
Subclasses	
Responsabilità	**Collaboratori**
Connessione al DB, prepare ed eseguire query per	*UtilityDB*

Nome Classe: *UtilityDB*	**Package**: DB
Superclasses	-
Subclasses	-

Responsabilità	**Collaboratori**
Metodi statici per la connessione e la chiusura di connessione relative alla basi di dati Oracle	**-**