**Documento dell’Architettura software**

|  |  |  |  |
| --- | --- | --- | --- |
| **Versione** | **Data** | **Descrizione** | **Autore** |
| Elaborazione 1 | 11/11/2013 | Inizializzazione | Luca finocchio  Vittoriano Muttillo  Daniele Leombruni  Stefano Dell’Osa |
| Elaborazione 2 | 09/12/2013 | Revisione | Luca finocchio  Vittoriano Muttillo  Daniele Leombruni  Stefano Dell’Osa |
| Elaborazione 3 | 13/01/2014 | Revisione | Luca finocchio  Vittoriano Muttillo  Daniele Leombruni  Stefano Dell’Osa |

L’architettura software dell’applicazione Time-Logging si basa su alcuni principi e pattern fondanti.

In particolare si è scelto di realizzare un’**architettura distribuita** e di strutturare il software implementando il pattern MVC.

Per quanto concerne l’architettura distribuita da noi implementata, si è scelto di realizzare un client “leggero” (solo interfaccia utente) ed un server robusto (logica applicativa, gestione dati, etc..).

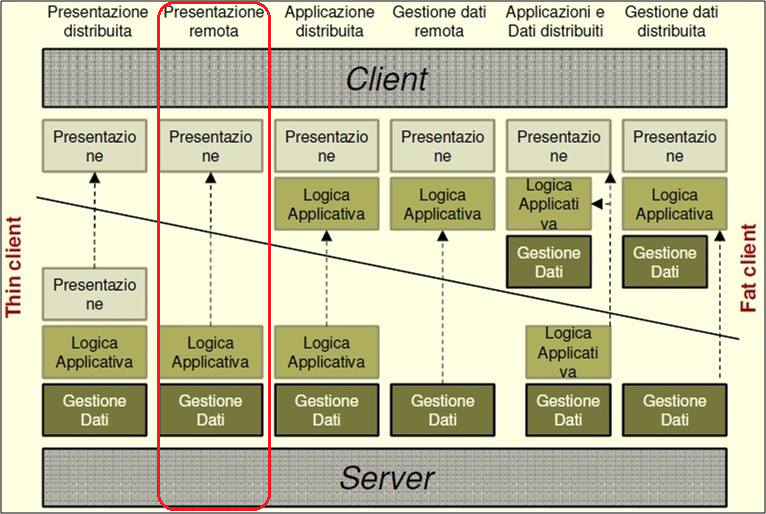
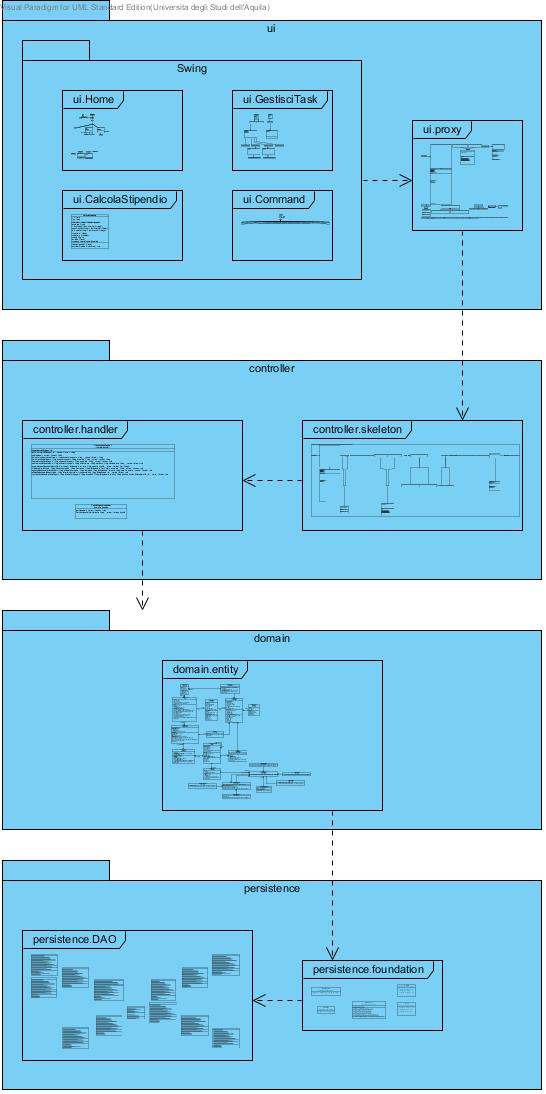


Figura - Architetture client server

La figura 1 di cui sopra ci aiuta a comprendere le possibili implementazioni di architetture distribuite ed in particolare nel riquadro in rosso troviamo quella da noi scelta per l’applicazione Time-Logging.

Una volta scelta l’architettura distribuita si è passati alla definizione di package e delle classi software che andranno a caratterizzare la nostra applicazione.

Il dettaglio della struttura dell’architettura software da noi realizzata si evince dall’immagine seguente:



**Server**: Si compone di tre package, rispettivamente:

* **Controller** : riceve i comandi dall’utente attraverso la **view** (UI sul Client) e li attua modificando lo stato del **Model** (Domain).
* **Domain** : fornisce gli strumenti per accedere ai dati utili all’applicazione.
* **Foundation :** fornisce strumenti per interfacciarsi con il DB tramite le classi del subpackage DAO.

**Client**: Le classi del subpackage **Swing**, demandate alla realizzazione dell’interfaccia, comunicano con il Server tramite le classi contenute nel subpackage **Proxy**

Figura – Diagramma dei package

Scendendo nel dettaglio della struttura dei package osserviamo la presenza di diversi subpackage, ognuno dei quali è stato creato per racchiudere le classi in base alle differenti funzionalità e ruoli che andranno a ad assolvere.

Ad esempio, i subpackage **Skeleton** e **Proxy** sono stati introdotti per implementare l’architettura distribuita seguendo le specifiche introdotte dal software ICE, da noi scelto per realizzare la distribuzione. In particolare essi sono generati in maniera automatica una volta definito uno *slice*.

***Slice*** *(Specification Language for Ice) is the fundamental abstraction mechanism for separating object interfaces from their implementations. Slice establishes a contract between client and server that describes the types and object interfaces used by an application. This description is independent of the implementation language, so it does not matter whether the client is written in the same language as the server.*

Ogni *slice* viene mappato sul **client** come classi **Proxy** e sul **Server** come classi **Skeleton**.

La presenza del subpackage **DAO** all’interno del package **foundation** è dovuto ad un ulteriore vincolo tecnologico è imposto dall’utilizzo di un ORM, nello specifico *Hibernate*.

Anche in questo caso, le classi contenute nel subpackage **DAO** sono state generate in maniera automatica da Hibernate stesso.

In aggiunta, al fine di favorire l’espandibilità e la duttilità del software si è scelto di implementare un nuovo strato, posizionato concettualmente al di sopra del DAO, il cui compito è quello di fornire un’ ”interfaccia” al model indipendente dalla tecnologia utilizzata per gestire la persistenza.

**Progettazione guidata dalle responsabilità (RDD)**

In conformità con l’approccio **RDD** (Responsbility-Driven Development), nella realizzazione dell’applicazione si è scelto di considerare gli oggetti software come dotati di responsabilità. Di seguito vedremo quindi quali pattern sono stati applicati in relazione alle problematiche sorte durante la fase di progettazione, ricordando che tali soluzioni sono state attentamente ponderate al fine di mantenere un basso accoppiamento e una coesione più alta possibile.

* La prima criticità che ci siamo trovati ad affrontare riguarda l’individuazione dell’oggetto, oltre lo strato UI, il cui compito è quello di ricevere e coordinare le operazioni di sistema.

La soluzione è stata trovata applicando i principi del **pattern GRASP Controller**. Si è scelto quindi di creare una nuova classe, rappresentante il caso d’uso, chiamata *gestisciTaskHandler* (come suggerito dal pattern) la quale si fa carico di ricevere le richieste provenienti da parte dell’utente (tramite l’interfaccia UI) e di gestirla.

* Un’altra criticità è rappresentata dalla necessità di individuare un principio base per l’assegnazione delle responsabilità agli oggetti. Relativamente al nostro contesto, occorreva individuare l’oggetto in grado di recuperare il *sotto-progetto* al quale verrà aggiunto/modificato/eliminato il *task*.

Una soluzione ci è stata fornita dal **pattern Information Expert**, il quale suggerisce di assegnare tale responsabilità alla classe che possiede tutte le informazioni necessarie per soddisfarla, nel nostro caso la classe *progetto*. Tale classe, infatti, “contiene” al suo interno i *sotto-progetti* di interesse. Per motivazioni analoghe si è scelto di assegnare alla classe Azienda la responsabilità di recuperare i *progetti*.

* La criticità relativa alla creazione di nuovi oggetti è stata superata avvalendosi del **pattern Creator**. In particolare, la responsabilità di creare/modificare/eliminare un nuovo oggetto t*ask* è stata assegnata alla classe *sotto-progetto* in quanto, come specificato anche dal pattern stesso, essa “contiene” oggetti di tipo Task, li utilizza strettamente e si occupa della loro memorizzazione. Inizialmente si era pensato di assegnare questa responsabilità direttamente alla classe *gestisciTaskHandler*, la quale possiede i dati per l’inizializzazione del Task. Tuttavia la prima strada è stata preferita a quest’ultima poiché consentiva di minimizzare il livello di accoppiamento (in questo modo *gestisciTaskHandler* non è accoppiato con Task).
* Un’ulteriore criticità riguardava il come gestire alternative basate sul tipo, ovvero come gestire situazioni in cui le alternative o i comportamenti correlati variano con il tipo (classe).

Nel nostro caso era necessario gestire la differenziazione dei dipendenti che lavorano ad un task, ognuno dei quali può avere caratteristiche diverse pur svolgendo lo stesso compito.

La soluzione a questo problema è stata trovata applicando il **pattern Polymorphism** il quale suggerisce di assegnare la responsabilità del comportamento ai tipi per i quali il comportamento varia, utilizzando operazioni *polimorfe*.

In particolare si è scelto di realizzare una classe astratta *Dipendente* a partire dalla quale implementare le classi specifiche per ciascuna tipologia di dipendente (Es. Consulente, Tirocinante, etc..)

* Utilizzando il pattern Information Expert ci si è posti il problema di gestire la bassa coesione che esso comporta, specie per la classe Azienda alla quale sono state assegnate le responsabilità di gestione sia dei *sotto-progetti* che dei *dipendenti*. Al fine di ottenere un’alta coesione (pur mantenendo un basso livello di accoppiamento ) si è scelto di applicare il pattern **Pure Fabrication,** il quale suggerisce di assegnare un insieme altamente coeso di responsabilità ad una classe artificiale o di convenienza che non rappresenta un concetto di dominio del problema. Nel nostro caso si è scelto di creare una classe *Direzione* alla quale è stata assegnata la responsabilità della classe *Dipendente.*
* Una criticità è stata quella di poter visualizzare diverse interfacce grafiche a seconda dell'utente che si trova ad utilizzare il software in un determinato momento (Azienda, Manager oppure generico Dipendente). In questo caso si è scelto di utilizzare il pattern **Abstract Factory** che ci permette di utilizzare elementi grafici differenti a seconda della tipologia di utente specifico senza cambiare lo standard grafico generale utilizzato. Nel caso specifico si è creata una classe interfaccia **ContentFactory** che definisce dei metodi factory per la creazione degli elementi grafici generici utilizzati (**createTreePanel** e **createPanel**) mentre la responsabilità della creazione effettiva delle generiche istanze degli elementi grafici appartiene alle sottoclassi che ridefiniscono i metodi dichiarati nell'interfaccia. Nel caso del Manager abbiamo una sottoclasse concreta **ContentManager** che customizzerà gli elementi presenti nel suo content (**createTreePanel** e **createPanel**) visualizzando solo l'interfaccia specifica creata per il Manager.
* La criticità relativa alla gestione delle regole per il calcolo dello stipendio è stata superata grazie all’utilizzo dei pattern **Strategy** e **Composite**. In particolare il pattern *Strategy* ci ha consentito di definire ciascuna delle strategie di calcolo dello stipendio in classi separate, tutte aventi un’interfaccia comune. Il pattern *Composite*, invece, ci ha consentito di considerare il calcolo dello stipendio come un insieme di componenti che contribuiscono al calcolo stesso.
* Una criticità individuata a livello di interfaccia grafica consisteva nella gestione delle azioni da compiere al verificarsi di un evento associato ad un determinato oggetto o componente dell’interfaccia. Questo poiché talvolta è necessario inoltrare richieste ad oggetti senza conoscere nulla dell’operazione richiesta o del destinatario della richiesta.

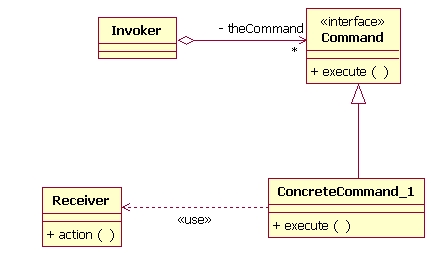


Figura - Diagramma delle classi del pattern Command

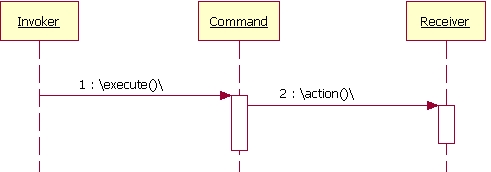


Figura - Sequence Diagram del pattern Command

A soluzione di tale criticità si è utilizzato il pattern **Command.** In particolare, si è deciso di implementare tale pattern avvalendosi della struttura già presente nelle librerie Swing di Java (vedi figura sotto).

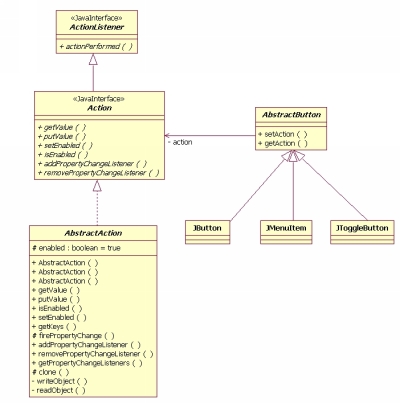


Figura - Diagramma delle classi "Swing Actions"