

**PROGETTO SAGRADA**



# **POLITECNICO**

## **MILANO 1863**

**Progetto di Ingegneria del Software 2017/2018**

**Autori:**

**Daniele Comi 10528029**

**Valerio Colombo 10499221**

**Valentina Deda 10488563**

## Sommario

Abstract.....	5
Introduzione.....	6
Analisi scelta DBMS.....	7
Perchè MySQL? .....	7
Analisi struttura concettuale database .....	8
Spiegazione modello concettuale .....	9
Disclaimer: implementazione effettiva .....	10
Analisi protocollo di scambio pacchetti messaggio/dati tra client e server in un'architettura client server .....	11
Analisi interazioni client server attraverso scambio di messaggi in formato JSON .....	14
Analisi protocollo di trasporto .....	17
Contromisure .....	18
Utilizzo del protocollo UDP .....	18
Descrizione protocollo Heartbeat .....	18
Analisi creazione server attraverso socket .....	19
Vantaggi.....	21
Utilizzo del design pattern MVC.....	22
Messaging Systems Dispatcher .....	23
Messages .....	23
Channels .....	23
Dynamic Routers.....	23
Implementare il Framework. ....	23
Implementare il Framework .....	24
Events .....	24
Handlers.....	24
Event Dispatcher .....	24
Gestione distribuita dei controller MVC .....	27
Gestione della sicurezza .....	30
Gestione della concorrenza.....	31
Struttura generale interna del progetto .....	32
Regole per lo sviluppo in team delle procedure Java .....	35
1.1    Introduzione .....	36

1.2	Organizzazione del codice .....	36
1.2.1	Struttura generale dell'ambiente operativo Java .....	36
1.2.2	Organizzazione di massima del codice sorgente.....	37
1.3	Convenzioni relative al linguaggio Java .....	39
1.4	Documentazione dei sorgenti.....	44
1.4.1	Documentazione a livello di classe .....	44
1.4.2	Documentazione a livello di metodo.....	45
1.4.3	Altre forme di documentazione.....	45
1.5	IDE, Testing, Repository e software ausiliari .....	46
1.5.1	IDE.....	46
1.5.2	Testing.....	46
1.5.3	Repository.....	46
1.5.4	Software ausiliari.....	47

## Abstract

Questo documento ha il compito di illustrare la fase progettuale ed implementativa del progetto di Ingegneria del Software. Il progetto consisteva nel realizzare la versione digitale di un gioco da tavolo chiamato "Sagrada". Qui di seguito una breve descrizione dello svolgimento del gioco per una maggiore chiarezza nella comprensione del gioco stesso:

*Siete artisti in competizione tra loro per creare la vetrata più spettacolare della Sagrada Familia. I vostri frammenti di vetro sono rappresentati da dadi, caratterizzati da un colore e da una sfumatura, che è indicata dal loro valore numerico (più basso il valore più la sfumatura è chiara). Ad ogni round, i giocatori a turno effettuano un draft da un mazzo di carte, piazzando i dadi sulle proprie finestre. I giocatori devono piazzare seguendo le restrizioni della propria Carta Schema e non possono piazzare dadi adiacenti con lo stesso colore o valore. Dopo 10 round, i giocatori ottengono punti in funzione degli obiettivi pubblici e privati che hanno raggiunto, il giocatore con più punti è il vincitore!*

## Introduzione

### Obbiettivi di sviluppo

- Costruire un insieme modulare e indipendente di diversi pacchetti, classi e interfacce collegate tra diversi e potenti design pattern.
- Costruire una forte interfaccia grafica modulare modellata per un videogioco, includendo quindi varie funzionalità disponibili in JavaFX
- Costruire una forte rete multi giocatore modulare, utilizzando sia socket che RMI
- Infrastruttura topologica di rete: three-tier client/server

### Strumenti di lavoro

In questo paragrafo verrà esposto brevemente, sotto forma di lista sintetica, l'elenco di strumenti e tecnologie di terzi utilizzate per l'agevolazione dello sviluppo; ne seguirà più avanti nel documento una migliore e più approfondita descrizione.

- **Tecnologie di sviluppo:**



- **Tecnologie per il testing e mantenimento del codice:**



# Gestione database in un'architettura three tier client server

## Analisi scelta DBMS

La scelta del Database Management System non è una scelta tanto banale quanto può sembrare. In questo progetto era originariamente stato scelto di utilizzare MySQL, quindi più precisamente un Relational Database Manangement System. Si vedano i **desclaimer** alla fine di questa sezione per un commento sull'implementazione effettiva contrapposta all'idea originale.

## Perchè MySQL?

### Sicurezza

MySQL utilizza una sicurezza distribuita su più livelli andando a sfruttare il già esistente protocollo che ne fa da garante, SSL. MySQL, supportandolo, permette quindi una comunicazione sicura e criptata attraverso certificati e cifratura con chiavi RSA da 2048 bit.

### Affidabilità

MySQL ha un'ottima gestione degli errori interni dovuti a un qualche tipo di corruzione del database riuscendo a svolgere eventuali operazioni di ripristino necessarie.

### Performance

Una un'istanza propriamente configurata di MySQL è molto veloce per i seguenti motivi:

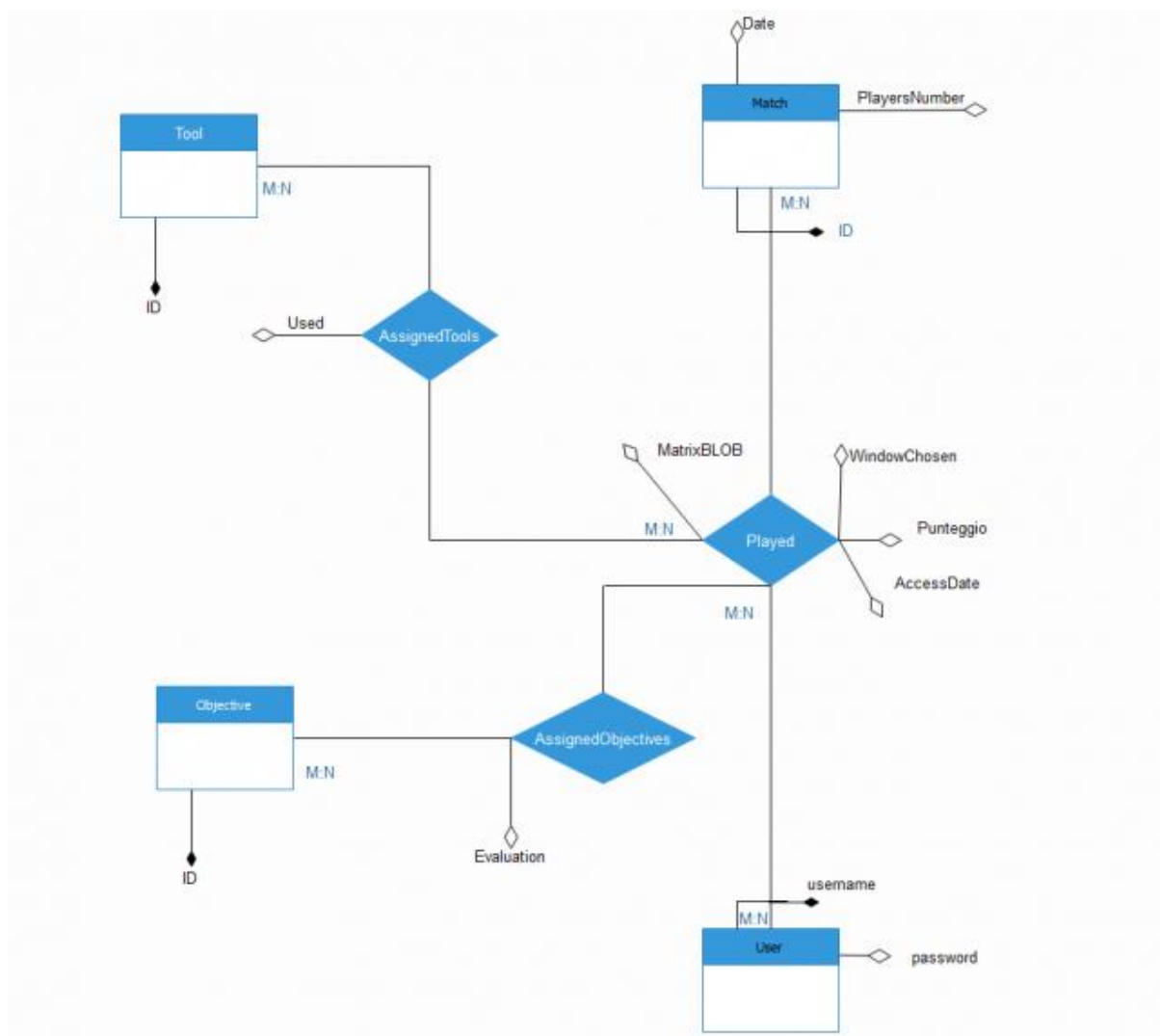
- Covering index support
- Adaptive hash index
- Buffer pool implementation
- Adaptive checkpointing
- Binary log replication

## Analisi struttura concettuale database

L'utilizzo di un database per la memorizzazione di vari tipi di dati, dagli utenti ai loro salvataggi alle loro partite e molto altro, è subito sembrato la scelta più ovvia per questo progetto.

In questo documento verrà esposto il modello relazionale che era stato scelto per la struttura interna del database e la consistenza dei dati in esso memorizzato.

L'UML nell'immagine sottostante indica come il database sarebbe stato implementato concettualmente





## Spiegazione modello concettuale

Si può notare la presenza di relazioni M:N, questo comporta la creazione di ulteriori tabelle per poter creare una corrispondenza ed una coerenza tra i dati in esso memorizzati. Scendendo ulteriormente nel dettaglio, si possono analizzare le scelte pensate per il modello concettuale che qui sotto verrà esposto in ogni suo punto.

La tabella User avrebbe contenuto 3 campi :

- username (alfanumerico e chiave primaria)
- password (memorizzando solo il suo hash generato da MD5)
- date(data d'iscrizione al sistema di gioco).

La chiave primaria username sarebbe stata parte della chiave composta presente nella tabella generata dalla relazione M:N tra User e Match.

Match avrebbe contenuto anch'esso tre campi :

- date(data della partita)
- PlayersNumber(numero giocatori presenti)
- ID (identificativo partita e chiave primaria)

Anche in questo caso la chiave primaria, questa volta di Match ovvero ID, sarebbe stata parte della chiave composta della tabella creata dalla relazione M:N tra Match e User.

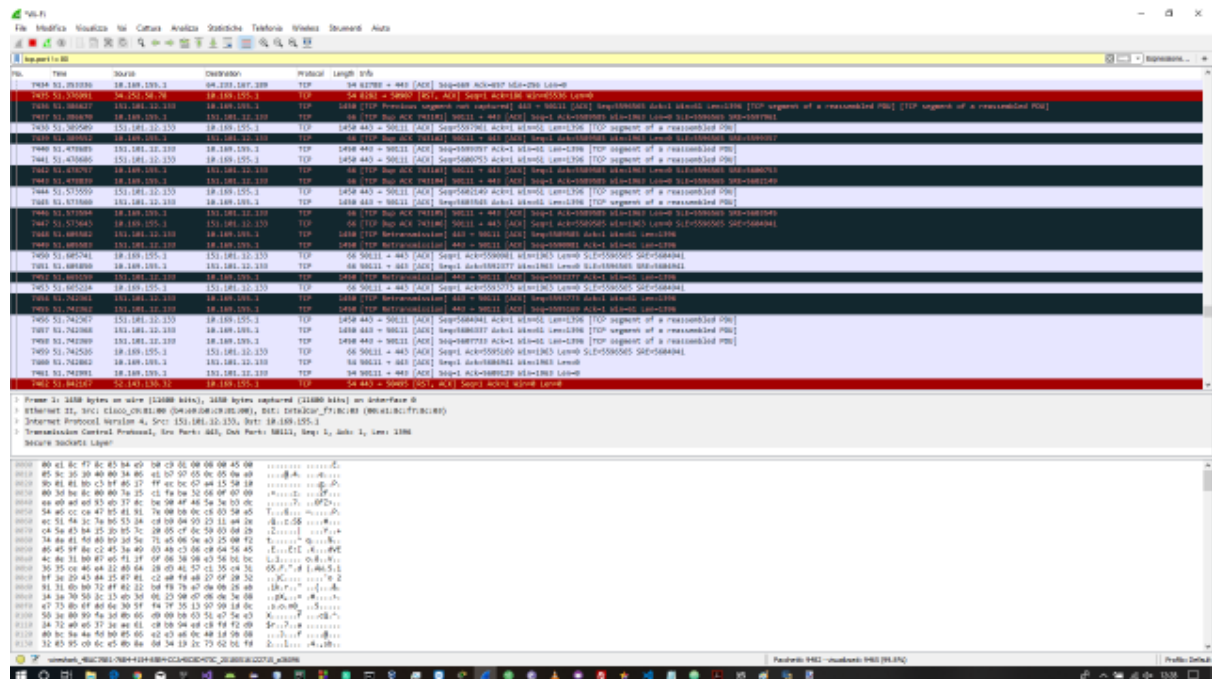
Questa tabella generata dalla relazione M:N doveva corrispondere al salvataggio/record di una partita per singolo giocatore ed essa conterrà :

- il punteggio raggiunto
- la data di accesso alla partita/sala d'attesa
- l'ID della vetrata scelta
- la rappresentazione sotto forma di BLOB (Binary Large Object) dello stato della vetrata a fine partita.

Inoltre sarebbero state memorizzate le informazioni sull'utilizzo e l'assegnamento di carte obbiettivo e strumento attraverso due ulteriori relazioni M:N tra la tabella generata dalla relazione tra le due precedenti entità (Match e User).

## Problemi riscontrati e soluzioni

La porta utilizzata per il server in cui sarebbe stato presente MySQL dovrebbe essere la standard 3306 in linea teorica (porta registrata a MySQL da IANA), ma per ragioni dovute ad un packet filtering presente all'interno della rete del Politecnico a livello di trasporto, non è possibile utilizzare porte destinazione diverse dalla 80 e dalla 443, ve ne è una dimostrazione nell'immagine sotto riportata relativa ad un packet capturing effettuato attraverso il software Wireshark dove, escludendo qualsiasi pacchetto con porta destinazione nell'header corrispondente a 80, si può vedere come gli unici altri pacchetti presenti sono pacchetti con porte destinazione uguale a 443.



La soluzione proposta è quindi di attenersi all'utilizzo di un MySQL server remoto su porta 80 o 443.

## Disclaimer: implementazione effettiva

Per via delle indicazioni sulla consegna e svolgimento dell'esecuzione del software finale, è stato scelta l'implementazione del database utilizzando la tecnologia messa a disposizione dal DBMS **SQLite**, che consistendo in un unico file garantisce ovviamente una maggiore portabilità come risorsa a discapito dei vantaggi sopracitati MySQL, vantaggi ovviamente in questo caso prettamente teorici, in quanto sarebbero sussistiti laddove si fosse andato a creare a creare un ambiente dove sono presenti centinaia di client in diversi match, il che sarebbe stato possibile ovviamente con l'adeguato hardware ed infrastruttura di rete.

# Analisi protocollo di scambio pacchetti messaggio/dati tra client e server in un'architettura client server

---

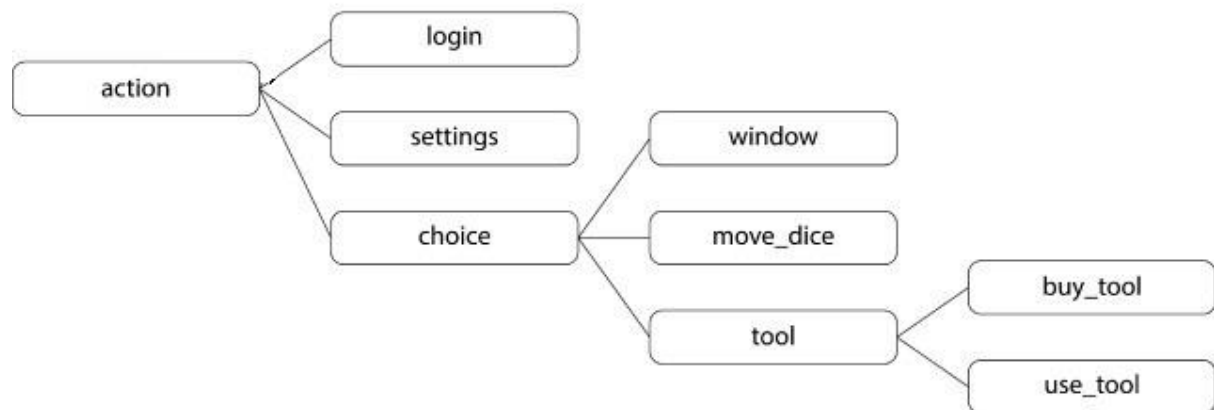
Server e client effettueranno uno scambio di messaggi attraverso il formato JSON. In particolare, ciascun messaggio è caratterizzato da *tre* elementi principali (elencati di seguito nell'ordine in cui appaiono in ogni messaggio): il *tipo di messaggio*, il *tipo di azione* e il *contenuto dell'azione*. Esiste anche un quarto elemento opzionale: il *nome dell'azione*.

In particolare, un generico messaggio avrà la seguente struttura:

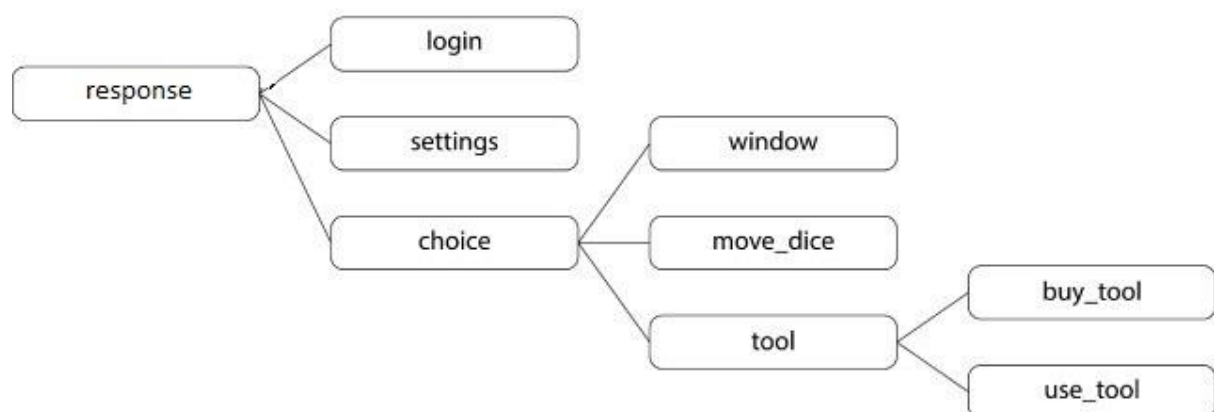
```
"tipo" : "tipo di messaggio",  
  "tipo di azione" : "tipo di azione del messaggio",  
  "[nome dell'azione]" : {  
    "contenuto dell'azione"  
  }  
}
```

Con "tipo di messaggio" si intende sinteticamente il genere di contenuto del messaggio: ad esempio, la parola chiave *action* indicherà che il messaggio contiene un'azione. Con "tipo di azione" si entra ulteriormente nel dettaglio: si indica infatti quale genere di azione, fra quelle disponibili, si intende compiere. Dunque la parola "login" indicherà la richiesta di registrazione, la parola "settings" indicherà il settaggio di alcune impostazioni di gioco, la parola "choice" indicherà un'azione di scelta fra varie opzioni. Fanno parte del "contenuto dell'azione" tutte le informazioni aggiuntive necessarie per portare a termine l'azione; ad esempio, in un messaggio di login questa sezione conterrà le credenziali dell'utente, mentre in un messaggio di scelta indicherà la scelta effettuata fra le opzioni disponibili, quali per esempio le coordinate a cui si intende posizionare un dado sulla Window. Il nome dell'azione, nel caso di un'azione specifica e identificata univocamente, indica il nome (che costituisce una breve descrizione dell'azione) attraverso il quale è possibile risalire all'azione stessa.

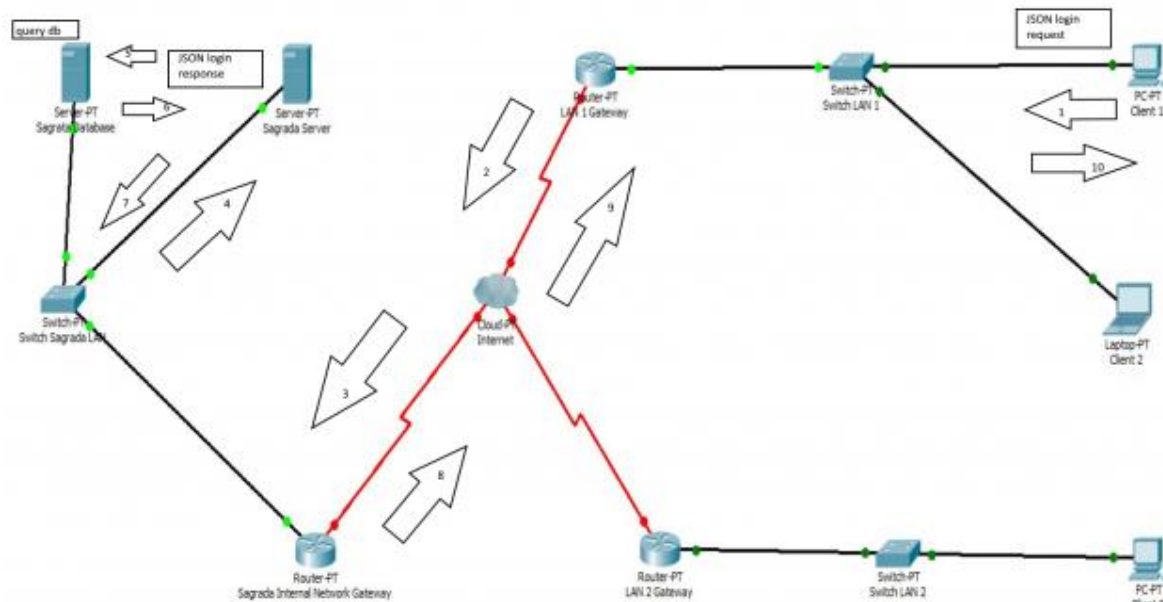
La struttura dei messaggi può essere visualizzata attraverso il seguente albero in riferimento al protocollo applicativo lato client



Qui invece si denota l'albero del protocollo applicativo lato server



Lo scambio di messaggi nell'architettura astratta di rete è possibile visualizzarla in una maniera più relativa all'infrastruttura di rete nel seguente grafico dove è possibile vedere lo scambio di pacchetti dati contenenti messaggi in JSON scambiati in questo caso durante l'interazione login tra un client ed il server.



# Analisi interazioni client server attraverso scambio di messaggi in formato JSON

---

Il server avviandosi creerà un servizio basato su TCP-IP in ascolto su una porta generata attraverso un protocollo di port discovery, al connettersi un certo client, oltre ad avviare il protocollo Heartbeat lato client al server via UDP, invierà un messaggio sotto forma di JSON contenente i dati per il login e ricevendo dal server una risposta, come da esempio. Questa richiesta di login contiene le credenziali del giocatore che si connette al server. Il server risponde con una conferma che segnala che il login è andato a buon fine.

## Client request

```
"type_msg" : "action",
"type_cmd" : "login",
"login" : {
  "username" : "username",
  "authentication" : "auth"
}
```

## Server response :

```
"type_msg" : "response",
"type_cmd" : "login",
"login" : {
  "valid_response" : "response",
  "metadata" : "metadata"
}
```

Successivamente, il client manda un messaggio in cui sono indicate le proprie scelte per quanto riguardano l'interfaccia di utilizzo (ovvero la scelta fra CLI e GUI). Anche la ricezione di questa richiesta viene confermata dal server.

## Client request :

```
"type_msg" : "action",
"type_cmd", "settings" : {
  "interface" : "gui",
}
```

**Server response :**

```

"type_msg" : "response",
"type_cmd" : "settings",
"settings" : {
    "valid_response" : "response",
    "metadata" : "metadata"
}
}

```

Il server poi inoltra al client gli ID delle due *Window Card* proposte: il client manda al server un messaggio che indica quale delle due *Window Card* intenda utilizzare e se della carta scelta intenda utilizzare il fronte o il retro. Dopo la scelta delle Window, il server invia al client l'ID della *Private Objective Card* a lui assegnata.

**Inoltro id della window al client:**

```

"type_msg" : "response",
"type_cmd" : "window_list",
"window_list" : {
    "id_player" : id_player,
    "window_id_1" : id1,
    "window_id_2" : id2
}
}

```

**Scelta iniziale window:**

```

"type_msg" : "action",
"type_cmd" : "choice_window",
>window" : {
    "id_player" : id_player
    "window_id" : "id",
    "window_side" : "side"
}
}

```

Durante il gioco verrà effettuato uno scambio di messaggi non dissimile dai precedenti, permettendo un'elevata modularità e dinamicità nella comunicazione tra Server e i vari Client, nonché tra le classi interne di un dato processo in esecuzione. Qui si riporta di seguito un esempio per ciascuna delle tipologie rilevanti di messaggio.

**Posizionamento di un dado nella vetrata :**

```

"type_msg" : "action",
"type_cmd" : "choice_move_dice",

```

```

"move_dice" : {
  "player_id" : player_id,
  "dice_id" : "dice_id",
  "source" : "source",
  "position" : {
    "x" : "x",
    "y" : "y"
  }
}
}

```

### Inoltro id dei dadi generati dal model :

```

"type_msg" : "response": {
  "type_cmd" : "dice_list",
  "move_dice" : {
    "destination" : "destination",
    "dice" : [
      {
        "dice_id_1" : "dice_id_1",
        "value" : "value",
        "color" : "color"
      },
      {
        "dice_id_1" : "dice_id_1",
        "value" : "value",
        "color" : "color"
      },
      ...
    ]
  }
}

```

### Acquisto di una toolcard :

```

"type_msg" : "action",
"type_cmd" : "choice_buy_toolcard",
"toolcard" : {
  "toolcard_id" : "toolcard_id"
}
}

```

### Risposta del server per l'acquisto di una toolcard :

```

"type_msg" : "response",
"type_cmd" : "choice_buy_toolcard",
"toolcard" : {
  "toolcard_id" : "toolcard_id",
  "purchasable" : "purchasable"
}
}

```

### Utilizzo toolcard :

```

"type_msg" : "action",

```



```

"type_cmd" : "choice_use_toolcard",
"toolcard" : {
  "toolcard_id" : "toolcard_id",
  "data" : {
    "position" : {
      "x" : "x",
      "y" : "y"
    },
    "other_data" : "other_data"
    ...
    ...
    ...
  }
}
}

```

### Risposta Server riguardo l'utilizzo di una toolcard :

```

"type_msg" : "response",
"type_cmd" : "choice_use_toolcard",
"toolcard" : {
  "toolcard_id" : "toolcard_id",
  "used" : "used"
}
}

```

## Analisi protocollo di trasporto

Per lo scambio di informazioni su stato e funzionamento basilare del sistema clients server vengono utilizzati entrambi i protocolli della pila protocollare ISO/OSI a livello 3, ovvero UDP e TCP.

### Utilizzo del protocollo TCP

Il protocollo TCP viene utilizzato per lo scambio dei dati di gioco e delle azioni che avvengono in base alle mosse dei client che avvengono nei loro applicativi. I canali TCP aperti saranno diversi:

- Canale TCP in ascolto per nuovi client in arrivo
- Canale TCP di una sala d'attesa e futura partita
- Vari canali TCP per i client connessi

## Contromisure

Per la gestione delle interruzioni di connessione verrà utilizzato un protocollo denominato Heartbeat, simile ad uno standard descritto dall'IEEE 1278. Esso permetterà di ascoltare attraverso il protocollo di trasporto UDP ping inviati dai client al server permettendo attraverso la creazione di eventi catturati da eventuali listener lo stato di un singolo client.

Una maggiore descrizione di tale protocollo è descritta successivamente

## Utilizzo del protocollo UDP

Il protocollo UDP viene utilizzato come detto in precedenza come appoggio per il protocollo Heartbeat controllando lo stato dei client.

## Descrizione protocollo Heartbeat

Il protocollo heartbeat è strutturato su una continua notifica di un datagramma UDP con un delay di 1000 millisecondi. Nel campo payload è contenuto l'ID identificativo dal client costituito originalmente dall'indirizzo MAC delle schede di rete in corrente utilizzo dalla sorgente del datagramma, ma successivamente sostituito dall'username identificativo ed univoco per ogni giocatore: permettendo in questo l'avvio di più client attraverso la medesima interfaccia di rete. Il server rimarrà in ascolto dei vari datagrammi smistandoli a seconda dell'identificatore e gestendo i vari stati che essi stessi andranno a definire. Ogni qualvolta il server noterà che dopo un delay di 1000 millisecondi non ha ricevuto alcun datagramma dal client notificherà attraverso un evento l'utilizzatore di questo protocollo della probabile perdita di connessione consigliando di limitare lo scambio di dati attraverso un protocollo connessione-dipendente come TCP. Se entro 5000 millisecondi si riceverà un datagramma con tale identificativo il client risulterà di nuovo connesso e verrà ripristinato il canale TCP di comunicazione. Altrimenti il client verrà considerato completamente offline. Tutto questo verrà notificato come sopra citato attraverso una gestione ad eventi come qui sotto d'esempio è mostrato:

```
@Override
public void onHeartbeat(HeartbeatEvent event) {
    System.out.println("Received heartbet from " + event.getSource() + " in " +
event.getTimeFromPreviousBeat() + " at " + event.getBeatTimeStamp());
}

@Override
public void onDeath(HeartbeatEvent event) {
    System.out.println(event.getSource() + " died after " +
event.getTimeFromPreviousBeat() + " at " + event.getBeatTimeStamp());
}

@Override
```

```

public void onLossCommunication(HeartbeatEvent event) {
    System.out.println("Communication lost of " + event.getSource() + " in " +
event.getTimeFromPreviousBeat() + " at " + event.getBeatTimeStamp());
}

@Override
public void onReacquiredCommunication(HeartbeatEvent event) {
    System.out.println("Communication reacquired of " + event.getSource() + " in
" + event.getTimeFromPreviousBeat() + " at " + event.getBeatTimeStamp());
}

@Override
public void onAcquiredCommunication(HeartbeatEvent event) {
    System.out.println(event.getSource() + " connected at " +
event.getBeatTimeStamp());
}

```

Il codice d'esempio sopra mostra la cattura dei vari eventi sugli stati di un client, sarà poi compito di chi implementa il listener la decisione su cosa fare alla cattura di un certo tipo di evento.

L'evento generato oltre a contenere il payload conterrà anche il delay passato rispetto al precedente con il suo stesso payload identificatore e il timestamp della corrente ricezione del ping.

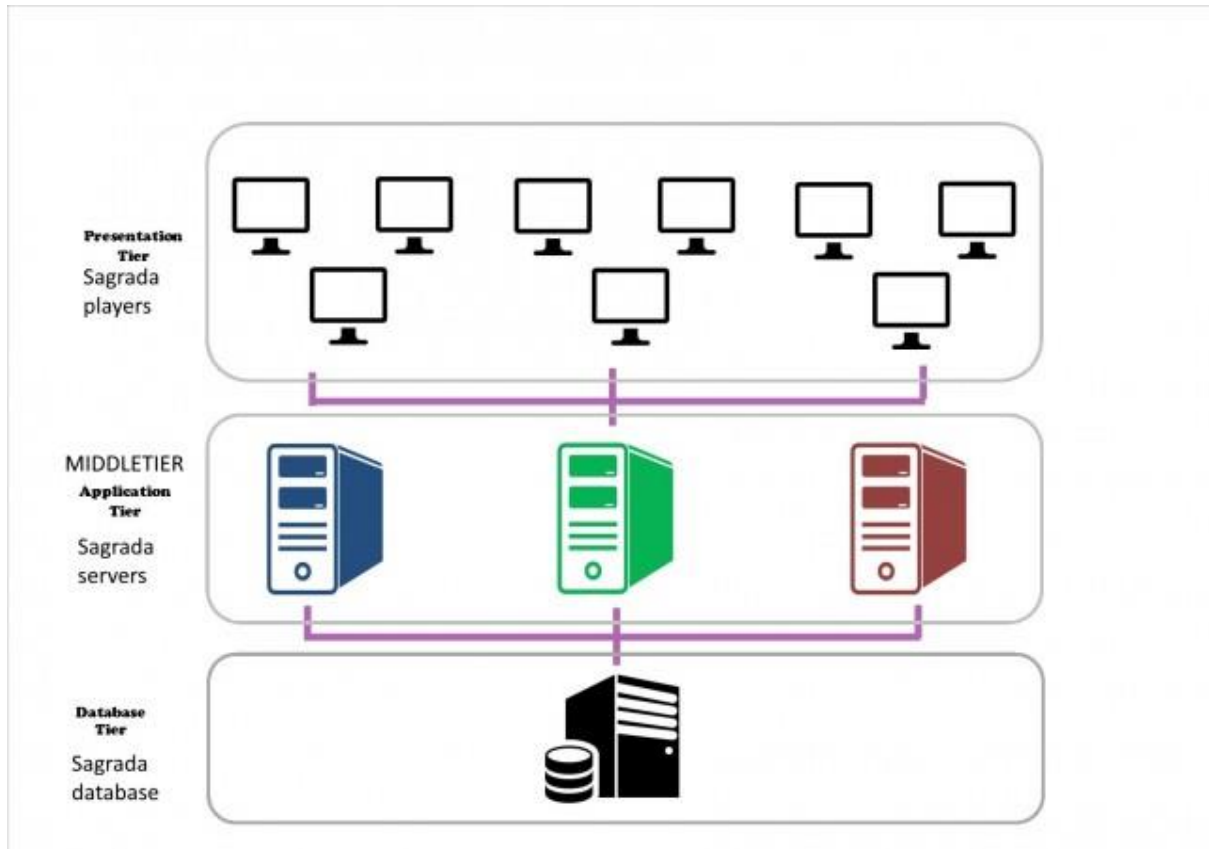
L'utilizzo di questo protocollo permetterà una migliore gestione dello stato delle connessioni tra client e server ed un maggiore controllo dei relativi errori possibili.

## Analisi creazione server attraverso socket

La creazione del server attraverso i socket porta una peculiarità da non sottovalutare, ovvero la scelta della porta. Non è possibile pensare di inizializzare un server in ascolto su una certa porta qualunque in un computer comune perchè comporta la possibilità di incorrere all'errore dovuto al corrente utilizzo di una porta da parte di un altro processo all'interno del sistema. Per evitare ciò potrebbe essere utilizzato un sistema dove attraverso le primitive TCP connect() permetteremo l'invio di un pacchetto SYN attendendo il tipo di risposta che verrà suddivisa in SYN|ACK e RST, il primo significherà una porta già in utilizzo mentre il secondo flag RESET ci indicherà l'impossibilità di apertura di una connessione e che quindi la porta è libera, una seconda tecnica potrebbe essere l'invio di un pacchetto di tipo FIN perchè accettato da un maggior numero di firewall. La risposta attesa sarà sempre un ACK (su FIN) o un RST. Per la scelta della porta in ascolto per il protocollo di trasporto UDP potrebbe essere utilizzato lo stesso sistema ma utilizzando la primitiva recvfrom(), inviando un generico payload di pochi byte si riceverà dall'host via UDP il flag ICMP\_PORT\_UNREACH. Il sistema utilizzato nel progetto è praticamente quanto sopra descritto ma lasciato fare il tutto attraverso le già codificate primitive e tecniche presenti quando avviene la chiamata alla primitiva di connessione sull'oggetto Socket.

# Scelta e analisi dell'architettura di rete

L'architettura di rete utilizzata a livello tipologico è l'architettura Client Server Three Tier. La tipologia è mostrata nella figura sottostante.



Tale architettura permette la suddivisione in 3 diversi livelli :

- Client tier: si occupa solo della parte di interazione con l'utente e visualizzazione grafica
- Middletier : corrispondente all'application layer, è il layer che comprende il server (o i server)
- Data tier : corrispondente al data layer, comprende il database (o i database)

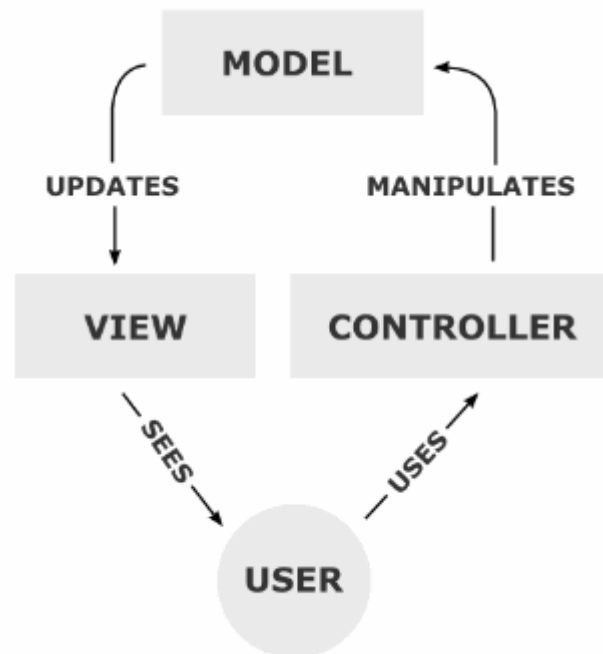
## Vantaggi

I vantaggi di questa architettura sono molteplici :

- Scalabilità : a discrezione della quantità di carico del sistema si può tranquillamente aumentare e quindi scalare il numero di server e/o di database server per garantire una migliore gestione del carico. Il bilanciamento del carico di lavoro sarà un compito gestito all'interno della LAN del middle tier, il client andando a fare una query DNS sull'hostname riceverà un certo indirizzo ip a cui poi si conatterà, all'interno della rete quando arriverà il primo pacchetto SYN|ACK dell'iniziale connessione TCP verrà gestito a quale server effettuare la connessione.
- Ridondanza : si ha sempre almeno un server, ai due livelli logici più alti, di riserva
- Gestione guasti : in qualsiasi momento ci sia un guasto in qualche componente di un singolo server, l'architettura intera di rete ridistribuirà il carico al meglio della nuova situazione lasciando inalterate le funzionalità del sistema di gioco, tutto in modo invisibile al client tier.

# Utilizzo del design pattern MVC

La struttura del design patter su cui questo progetto si poggia può essere visualizzata attraverso questa immagine.



Come si evince dal modello proposto la struttura del progetto prevede un ruolo centrale per il Controller che è ciò che permette all'utente di ricevere inviare richieste per delle azioni di necessario compimento come query, aggiornamenti sullo stato del gioco e molto altro come già visto poco prima nella struttura dei messaggi in formato JSON.

Il controllore è ciò che anche permette all'utente di conseguenza di apportare modifiche al model che una volta fatte potrà a seconda dei casi inviare dei messaggi all'utente che costituiranno ordini per l'aggiornamento dei vari componenti grafici presenti in base allo stato corrente ed aggiornato del model.

Nel progettare il controller si è deciso di rendere il controller una struttura distribuita piuttosto che centralizzata. Ovvero i vari messaggi di richiesta dati o di esecuzioni d'azioni provenienti dal client sono, prima di essere passati ai vari Controller, distribuiti attraverso un precedente layer costituito da un framework di Message System Dispatcher. Questo sistema verrà esposto di seguito prima di passare alla spiegazione della gestione distribuita dei controller.

## Messaging Systems Dispatcher

L'architettura basata sugli eventi segue diversi schemi dei sistemi di messaggistica. Si consideri l'analogo: eventi per i messaggi, gestori di eventi per canali e dispatcher di eventi ai router.

### Messages

Ricorda che l'analogo per un evento in un sistema di messaggistica è un messaggio. Ogni messaggio ha un tipo specificato che verrà utilizzato per associarsi a un gestore. Possiamo definire l'interfaccia per un messaggio come tale:

```
public interface Message {
  public Class<? extends Message> getType();
}
```

### Channels

Il secondo aspetto di un sistema basato sulla messaggistica è il punto di consegna. Dato che abbiamo una serie di messaggi, dobbiamo definire i punti di consegna che chiameremo canali.

Ogni canale sarà responsabile di uno o più tipi di messaggio; di conseguenza, possiamo inviare messaggi al rispettivo canale per l'elaborazione.

Possiamo quindi definire l'interfaccia per un canale:

```
public interface Channel<E extends Message> {
  public void dispatch(E message);
}
```

### Dynamic Routers

La gestione dei sistemi di messaggistica avviene attraverso i suoi router.

Il router si prende carico di instradare i vari messaggi verso i Channel che hanno effettuato un'iscrizione sul tipo del messaggio ricevuto.

La nostra interfaccia del router:

```
public interface DynamicRouter<E extends Message> {
  public void registerChannel(Class<? extends E> contentType,
    Channel<? extends E> channel);
  public abstract void dispatch(E content);
}
```

Tutto ciò che resta ora è implementare ciascuna delle interfacce per un framework completo.

## Implementare il Framework

### Events

Gli eventi sono dei veri e propri messaggi. Notando che ogni tipo di evento è identificato dalla sua classe, possiamo utilizzarla per definire univocamente l'evento al momento dell'instradamento all'interno del router.

```
import Message;

public class Event implements Message {
    @Override
    public Class<? extends Message> getType() {
        return getClass();
    }
}
```

### Handlers

I gestori di eventi fungono da punti di destinazione e di invio dei vari Message; quindi, implementiamo l'interfaccia del canale:

```
import Channel;

public class Handler implements Channel<Event> {
    @Override
    public void dispatch(Event message) {
        System.out.println(message.getClass());
    }
}
```

### Event Dispatcher

Il dispatcher ha il compito di registrare le iscrizioni dei canali ai messaggi.

Usiamo la HashMap nativa per associare eventi ai rispettivi gestori. Successivamente, possiamo semplicemente interrogare la mappa per memorizzare nuove associazioni (eventi, gestori) o per inviare eventi al gestore corretto.

```
public class EventDispatcher implements DynamicRouter<Event> {

    private Map<Class<? extends Event>, Handler> handlers;

    public EventDispatcher() {
        handlers = new HashMap<Class<? extends Event>, Handler>();
    }
}
```



```

@Override
public void registerChannel(Class<? extends Event> contentType,
    Channel<? extends Event> channel) {
    handlers.put(contentType, (Handler)channel);
}

@Override
public void dispatch(Event content) {
    handlers.get(content.getClass()).dispatch(content);
}
}

```

Ora il Framework è completo, qui un caso d'uso:

```

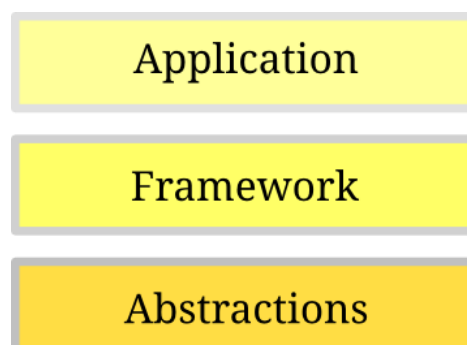
public class Program {
    public static void main(String[] args) {
        EventDispatcher dispatcher = new EventDispatcher();
        dispatcher.registerChannel(Event.class, new Handler());
        dispatcher.dispatch(new Event());
    }
}

```

Il codice sopra dovrebbe mostrare il nome della classe dell'evento nel terminale. Ora, tutte le applicazioni possono semplicemente registrare ulteriori eventi e gestori e inviarli.

Le classi Handler ed Event fanno parte del nostro livello framework che non devono essere modificati. Tutto il codice che estende il nostro framework esisterà sul livello d'applicazione. Il dovere del nostro framework è di astrarre e impalcare i componenti architettonici della nostra architettura basata sugli eventi.

È facile vedere ora che possiamo ereditare dalla classe Handler per creare i nostri gestori di eventi e che possiamo anche ereditare dalla classe Event per creare i nostri eventi. Inoltre, possiamo registrare questi gestori di eventi ed eventi per spedirli di conseguenza. Qui una visualizzazione di come questo Framework di messagistica si interfaccia all'interno del progetto:



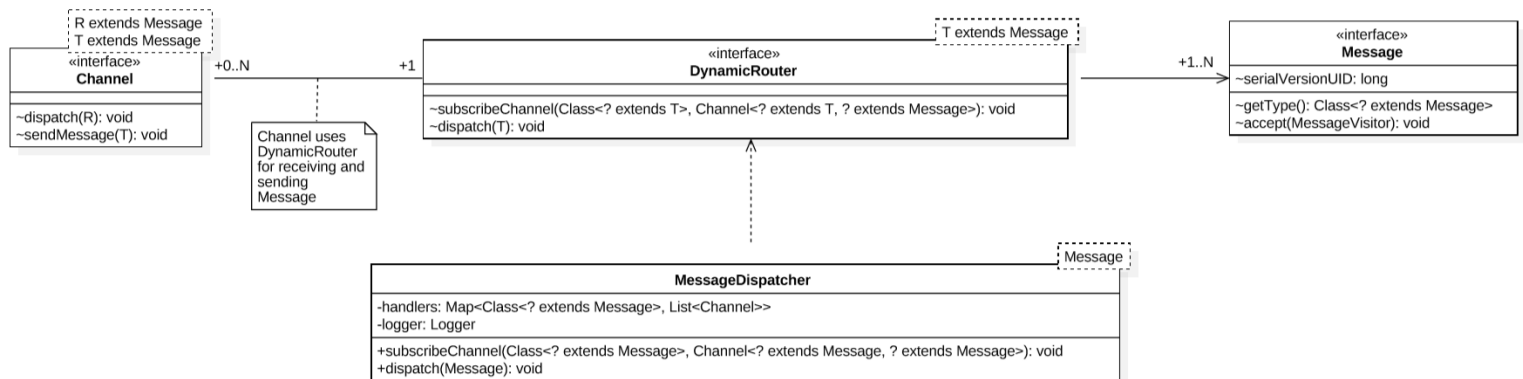
Per facilità di comprensione consideriamo il seguente esempio: Il Channel A vuole mandare un Message M al Channel B

Le fasi tipiche che si intraprendono sono le seguenti:

- B si iscrive, attraverso una implementazione del DynamicRouter, alla ricezione dei Message di tipo M
- A manda attraverso i metodi dell'interfaccia Channel un Message
- Il messaggio viene intercettato da una implementazione del DynamicRouter
- Il messaggio viene smistato in base al suo tipo verso tutti i Channel che si sono iscritti su quel tipo di messaggio

Per facilitare l'implementazione e la estensibilità del sistema il DynamicRouter utilizza le wildcard, questo permette ai vari Channel di iscriversi a più tipi di Message attraverso la stessa interfaccia fornita dal DynamicRouter. Questo permette, nel caso di questa applicazione, di instradare in modo dinamico i messaggi all'interno del model.

Di seguito il class diagram del Messaging Systems

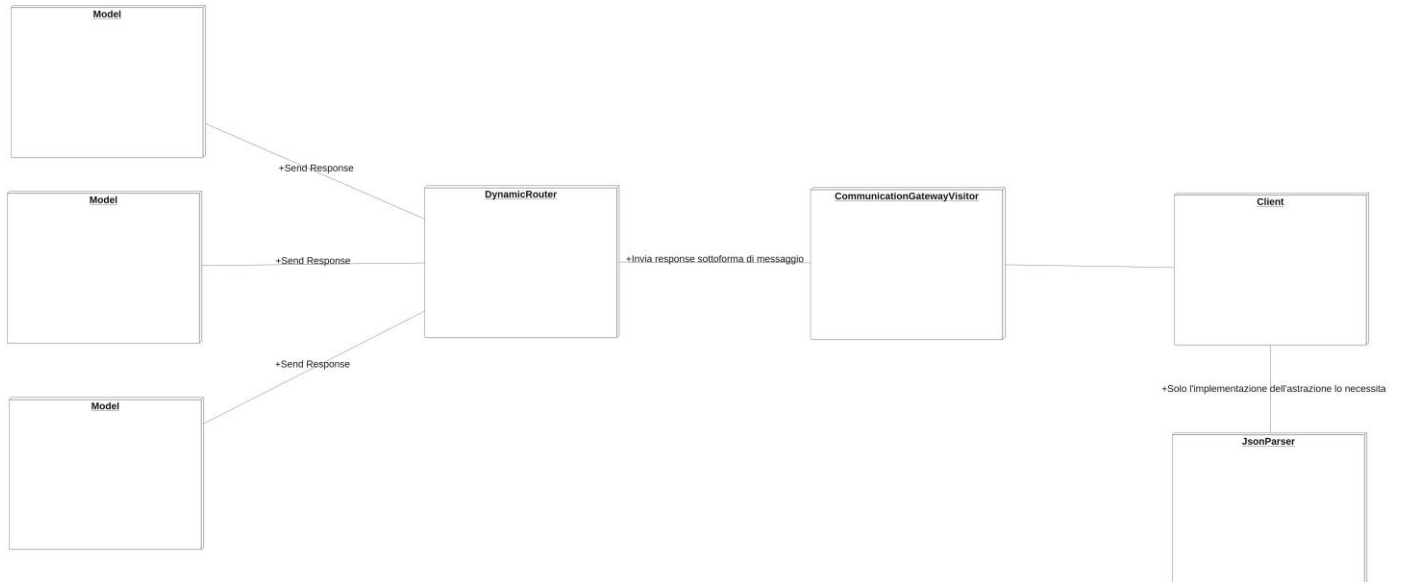


# Gestione distribuita dei controller MVC

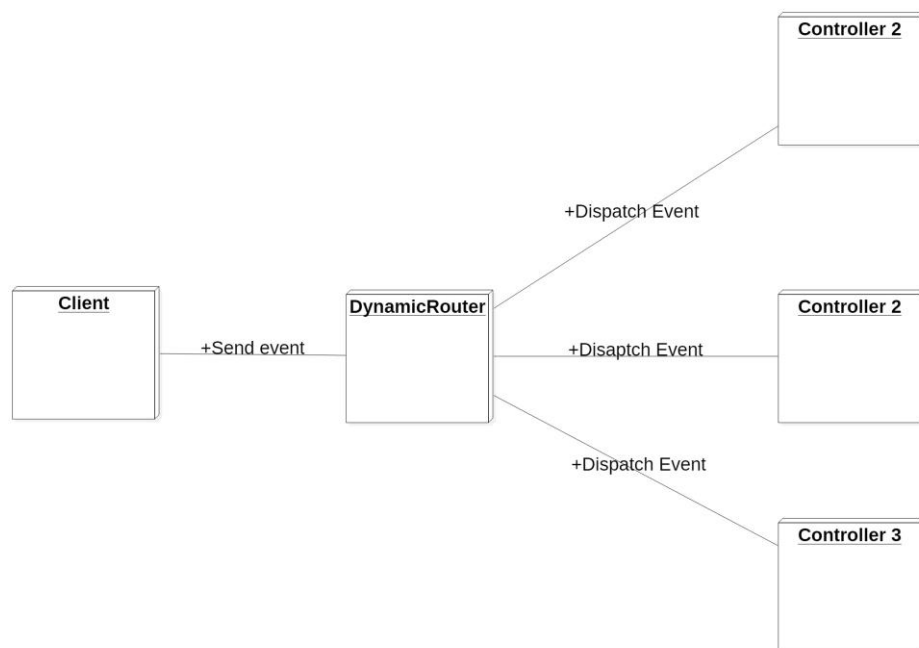
Tornando in merito alla struttura del design pattern MVC, è importante denotare la struttura distribuita dei controller ovvero che non ci sarà un controller centralizzato dove tutti gli eventi generati dal client verranno raccolti ma, dopo essere arrivati attraverso l'astrazione di rete, che più avanti verrà esaminata, verranno passati al framework del Message System Dispatcher che si occuperà di distribuire l'evento al controller specifico. Per maggiore chiarezza ecco un esempio:

Presupponendo che il client effettui una richiesta di posizionamento di un dado nella sua vetrata, una volta arrivato il messaggio esso verrà distribuito al manager competente che è in questo caso è la classe DiceManager. Essa oltre ad occuparsi della gestione dei dadi nel model sarà il punto d'arrivo per la ricezione dell'evento di un nuovo posizionamento di un dado. All'arrivo di questo evento il manager soddisferà la richiesta di elaborazione del messaggio.

Ora verrà mostrato, attraverso degli schemi richiamanti un Object Diagram, come avviene la comunicazione tra i due casi di ricezione e invio di un evento dal punto di vista del Server.



Nel caso di invio di un evento dal Server al Client, esso raggiunge il Client, o meglio l'implementazione della classe astratta Client, attraverso l'utilizzo del DynamicRouter. Un Channel all'interno del model manda il messaggio al dispatcher che lo incanala verso il Client il quale a sua volta attraverso il pattern Visitor utilizzerà il giusto metodo per interpretare il messaggio in entrata. Si fa notare la presenza di un Parser JSON che permette il parsing del messaggio sotto forma di contenuto testuale UTF-8 rappresentante una data struttura JSON descritta prima.

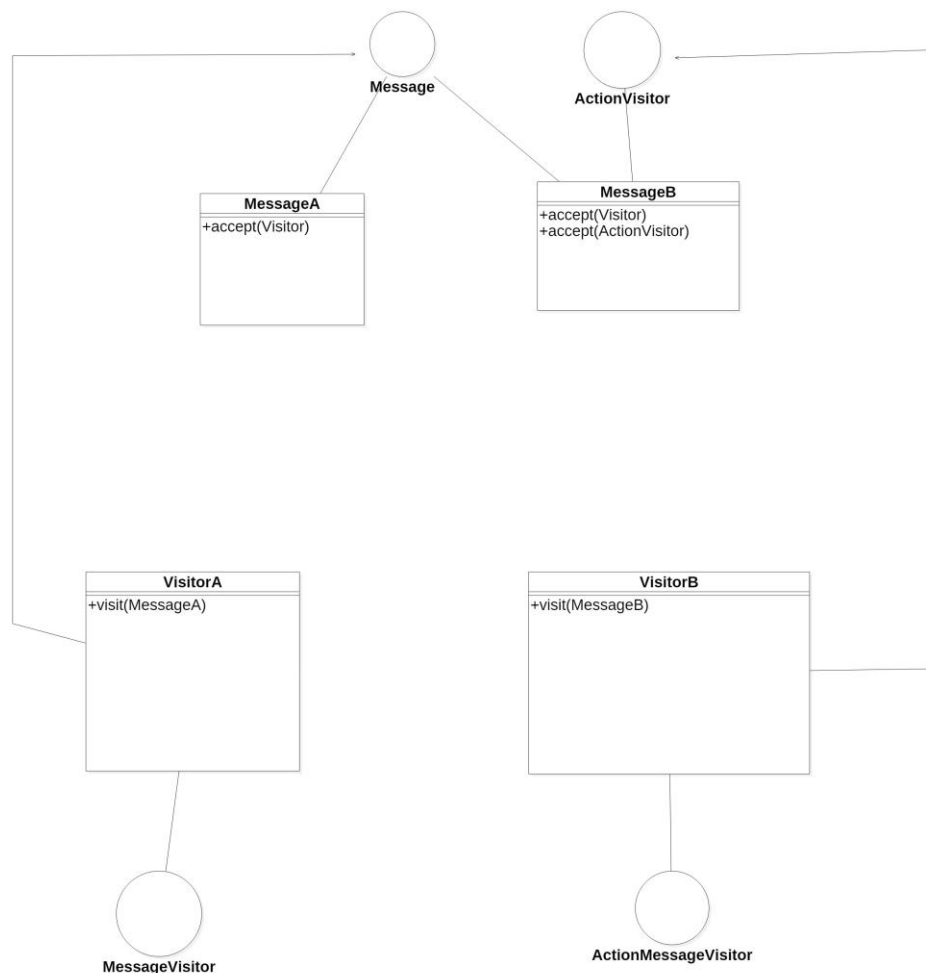


Il caso opposto è prettamente simmetrico. Il client invia attraverso la rete un Message il quale in entrata al server verrà intercettato dal DynamicRouter il quale instrada il messaggio verso i Channel che richiedono la ricezione di quel tipo di messaggio.

# Gestione messaggi ed eventi

E' di importanza assoluta un'ottima gestione dello scambio di messaggi ed eventi in invio e ricezione. Una parte è già stata affrontata analizzando il framework del Message System Dispatcher, a questo punto non rimane che identificare il tipo di messaggio in arrivo da parte del Dynamic Router. Questo è necessario per via di come i vari messaggi siano ovviamente in relazione d'ereditarietà con una generica classe messaggio.

Il riconoscimento è stato implementato attraverso un Visitor design pattern. Con questo approccio si è in grado di distinguere tra le varie istanze attraverso l'overloading dei vari metodi del design pattern Visitor. Ogni messaggio implementerà a seconda del suo specifico tipo, oltre all'interfaccia messaggio che già fornisce l'accesso all'utilizzatore Visitor, altre interfacce specifiche a seconda del tipo



## Gestione della sicurezza

Per garantire un minimo livello di sicurezza nell'interscambio di messaggi e l'eventuale salvataggio di dati si è predisposto l'utilizzo di alcuni algoritmi per la cifratura, decifratura ed hashing.

Per quanto concerne il Database che corrisponde al primo tier dell'architettura di rete, viene utilizzato l'algoritmo di hashing MD5 per lo storage delle password durante l'iscrizione di un utente al sistema.

Per quanto concerne invece la comunicazione di rete attraverso i socket è stata implementata l'interfaccia d'utilizzo per AES a 128 bit, permettendo in questo uno scambio sicuro di messaggi. Nonostante la chiave sincrona non venga scambiata durante alcun momento della creazione dell'astrazione della connessione, essa rimane la medesima in ogni momento insieme all'Initialization Vector. Questo perché è solo a scopo dimostrativo, citando il fatto che si può con facilità aggiornare il tutto utilizzando i SecureSocket messi a disposizione dal JDK per godere dell'utilizzo del protocollo SSL.

La chiave e l'Initialization Vector vengono memorizzati all'interno del JAR dove verranno prelevati all'instaurazione di una nuova cifratura o decifratura di un messaggio in invio o in arrivo.

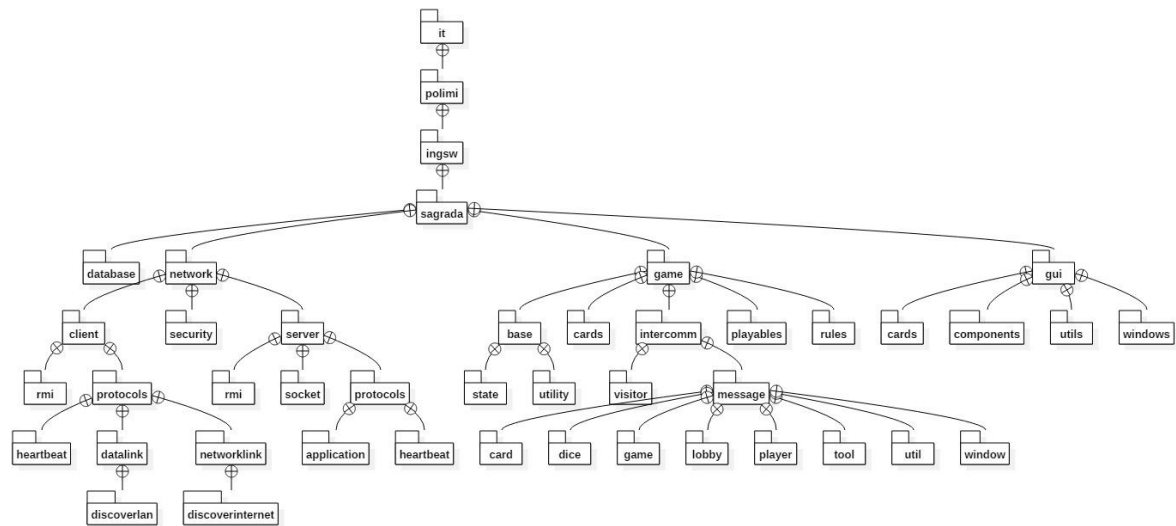
## Gestione della concorrenza

In questo progetto è stata scelta implementativa il rendere i servizi principali del server di gioco come serializzate, attraverso una struttura simil macchina a stati finiti, ovvero facendo in modo che ad ogni richiesta fatta corrisponda un unico flusso di dati principale e facendo in modo che ogni altra richiesta da parte dei client che non possano compiere azioni, per esempio, venga resa impossibilitata così da garantire una corretta esecuzione dell'intero gioco.

La gestione dell'accesso ad una lobby è avvenuta seguendo una logica praticamente simile andando a rendere ogni richiesta d'accesso una richiesta in attesa in una pila, garantita dalle richieste bloccanti di nuove connessione via socket TCP.

Oltre l'utilizzo della keyword `synchronized` ove richiesto per situazioni più particolari è stato necessario e comodo l'utilizzo di semafori per garantire un corretto ordine dell'inizializzazione della parte grafica del client nonostante i vari messaggi che arrivando potrebbero far sovrapporre più richieste non garantendo l'ordine di esecuzione corretto previsto.

# Struttura generale interna del progetto



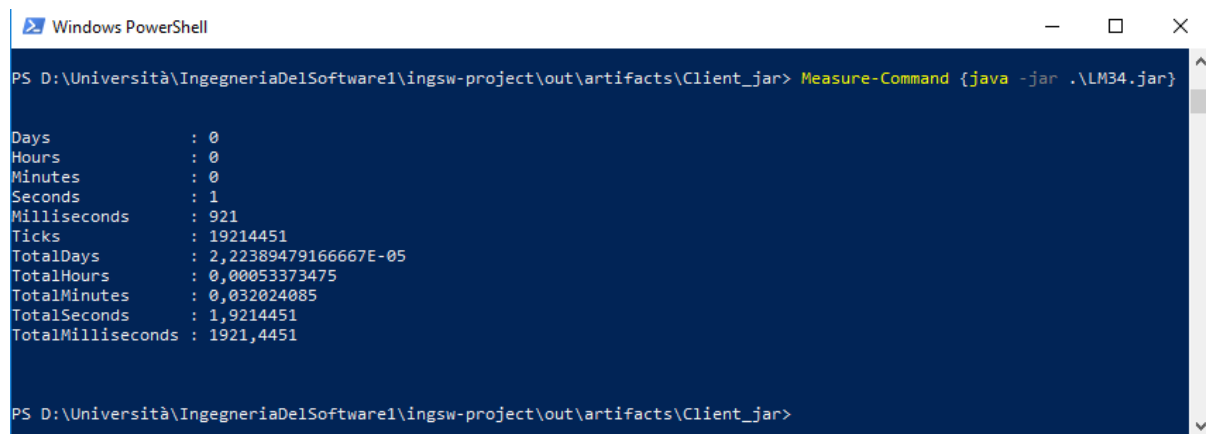


# Ottimizzazione post sviluppo

Successivamente al termine dello sviluppo si sono volute vanificare alcune opzioni per pensare a come ottimizzare il software, ovvero a renderlo il più performante possibile già all'avvio stesso. Si è presto capito che avremmo dovuto in qualche modo eludere la natura stessa di Java e del suo interpretare istruzione per istruzione al momento, ovvero in pratica il suo per così dire JIT all'interno della JVM. Per questo motivo si è provato ad utilizzare il software Excelsior Jet. Oltre a permetterci queste migliorie sotto ambiente Microsoft Windows, si è potuto creare anche un pacchetto di installazione di Sagra da permettendo così la sua installazione come un normale software qualunque.

Dopo aver provato a creare il pacchetto eseguibile, che dopo l'installazione l'eseguibile di gioco soltanto aveva un peso di ben più di 90 MB rispetto ai 19 originali del JAR, dovuti alla presenza "pre compilata" di Java SE, si è provato ad effettuare un qualche test di performance all'avvio stesso del gioco. Ecco di seguito i risultati ottenuti:

## Tempo di avvio del client JAR



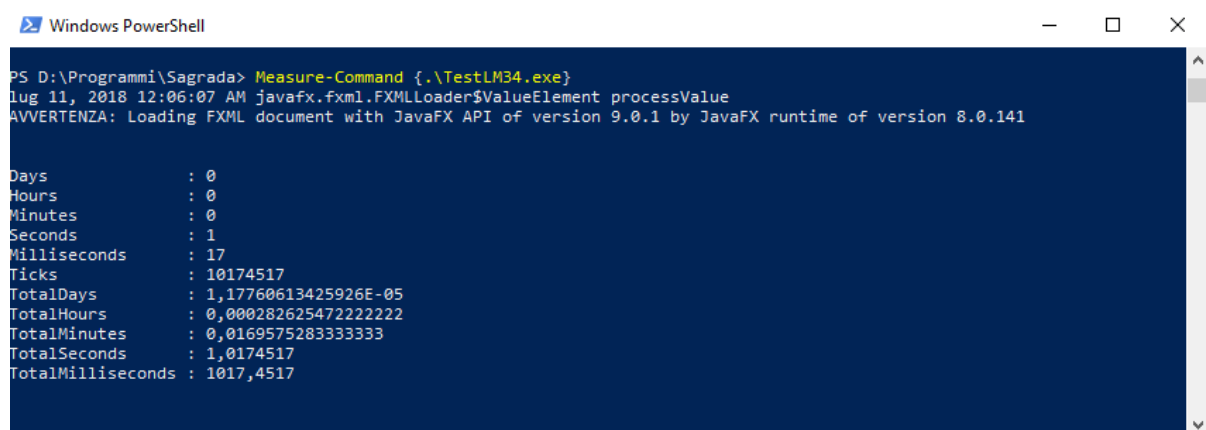
```

Windows PowerShell
PS D:\Università\IngegneriaDelSoftware\ingsw-project\out\artifacts\Client_jar> Measure-Command {java -jar .\LM34.jar}

Days           : 0
Hours          : 0
Minutes        : 0
Seconds        : 1
Milliseconds    : 921
Ticks          : 19214451
TotalDays      : 2,22389479166667E-05
TotalHours     : 0,00053373475
TotalMinutes   : 0,032024085
TotalSeconds   : 1,9214451
TotalMilliseconds : 1921,4451

PS D:\Università\IngegneriaDelSoftware\ingsw-project\out\artifacts\Client_jar>
  
```

## Tempo di avvio del client EXE

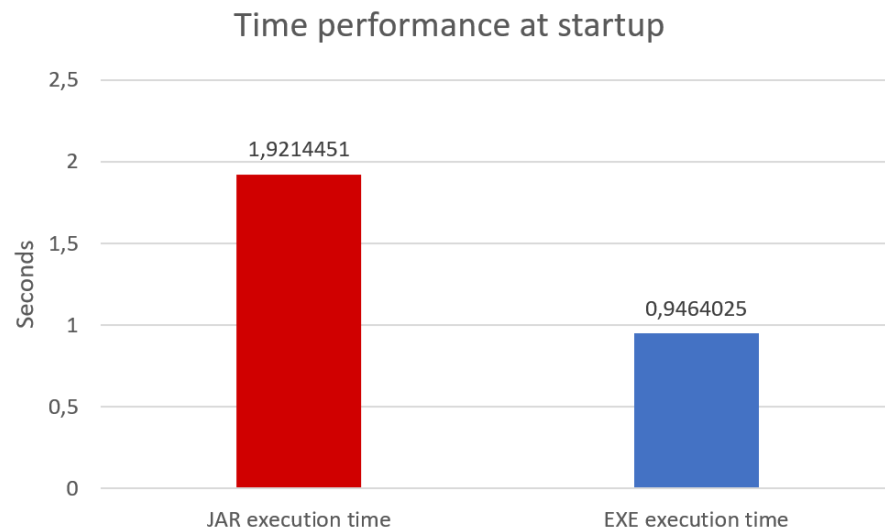


```

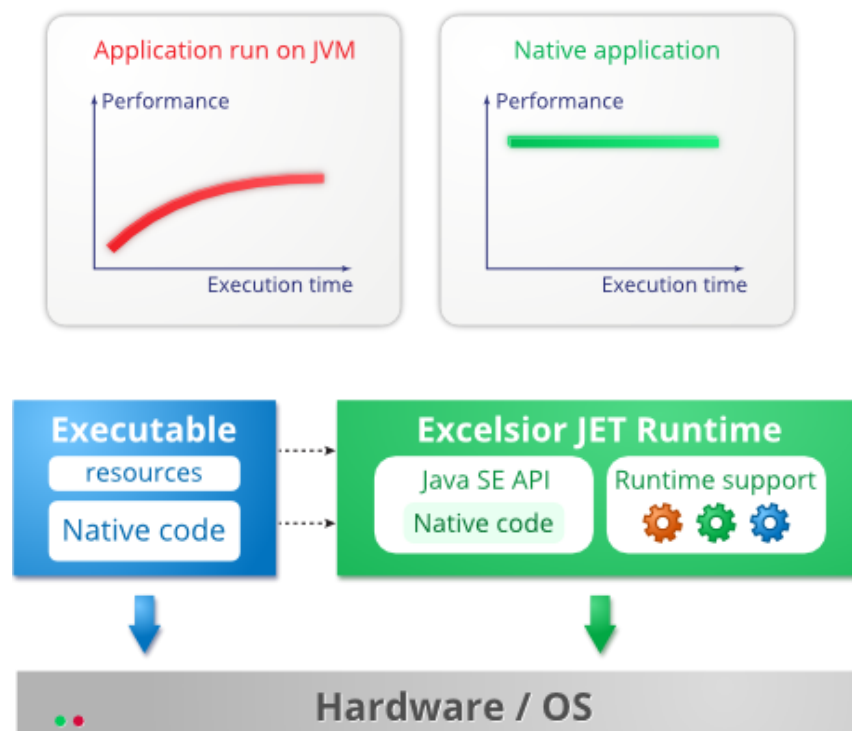
Windows PowerShell
PS D:\Programmi\Sagra> Measure-Command {.\TestLM34.exe}
Aug 11, 2018 12:06:07 AM javafx.fxml.FXMLLoader$ValueElement processValue
AVVERTENZA: Loading FXML document with JavaFX API of version 9.0.1 by JavaFX runtime of version 8.0.141

Days           : 0
Hours          : 0
Minutes        : 0
Seconds        : 1
Milliseconds    : 17
Ticks          : 10174517
TotalDays      : 1,17760613425926E-05
TotalHours     : 0,000282625472222222
TotalMinutes   : 0,0169575283333333
TotalSeconds   : 1,0174517
TotalMilliseconds : 1017,4517
  
```

### Tabella confronto tempi tra esecuzione JAR e EXE contenente una versione precompilata di Java 8 SE



I grafici mostrati dal software Excelsior Jet mostrano appunto come Java necessiti ad ogni avvio iniziale un certo tempo di "warm up", a differenza di un eseguibile per così dire "nativo" non necessiti di un certo tempo per raggiungere le migliori performance che potrebbero in ogni essere poco superiori delle prestazioni ottimali raggiunte da Java nel tempo. Mentre il successivo mostra come è strutturato internamente l'eseguibile convertito da JAR e di come sia ovviamente ancora strettamente legato a Java SE.



# **Regole per lo sviluppo in team delle procedure Java**

**Autore:** Daniele Comi  
**Data stesura:** 12 aprile 2018

## 1.1 Introduzione

Scopo del presente documento è quello di riassumere le regole generali da adottare nello sviluppo delle procedure Java. L'obiettivo è quello di identificare delle procedure standard che rendano uniforme la procedura di sviluppo e consentano la cooperazione tra diversi sviluppatori sullo stesso progetto.

## 1.2 Organizzazione del codice

Il codice Java prodotto deve essere ordinato utilizzando in maniera opportuna la strutturazione a *package* tipica del linguaggio Java. La dislocazione del codice è prefissata per consentire l'uniformità tra i vari ambienti di sviluppo e per permettere l'utilizzo di uno strumento comune di *versioning*.

### 1.2.1 Struttura generale dell'ambiente operativo Java

Tutto lo sviluppo Java relativo ai progetti in corso d'opera deve essere svolto all'interno di una struttura a directory ben definita. Più precisamente, l'ambiente di sviluppo deve avere la seguente struttura:

```

|__ Sagrada
    |__ .idea
    |
    |__ .scannerwork
    |
    |__ bin
    |
    |__ libs
    |
    |__ src
    |
    |__ .gitignore
    |
    |__ pom.xml
    |
    |__ sonar-project.properties
  
```

In dettaglio:

- **.idea:** directory contenente le varie personalizzazioni e impostazione relative all'IDE in utilizzo: IntelliJ IDEA 2018.

- **.scannerwork:** directory contenente impostazione e log relative alle precedenti e correnti revisioni automatizzate dei sorgenti del progetto Sagrada attraverso il software SonarQube.
- **bin:** directory che raccoglie le compilazioni, cioè tutti i file \*.class generati dalla compilazione dei sorgenti. Unitamente ai compilati deve contenere tutto il necessario al corretto funzionamento del software sviluppato. Questa directory sarà la base per il rilascio della release candidate ed eventualmente la final-stable release.
- Questa directory **non** contiene sorgenti quindi non è sottoposta a controllo di versione.
- **libs:** directory che raccoglie tutte le librerie necessarie allo sviluppo Java. Le librerie devono preferibilmente essere in formato compresso \*.jar o \*.zip; evitare il più possibile l'utilizzo di librerie basate su file system. Unitamente alle librerie questa directory include anche un file di testo UTF-8 che elenca le librerie contenute nella directory e alcune informazioni legate alle versioni e alla data di aggiornamento. Ogni qual volta si aggiunge una nuova libreria è necessario modificare questo file di testo ed aggiungere le informazioni relative alla nuova libreria introdotta.
- **src:** directory in cui sono presenti tutti i sorgenti del progetto Sagrada, raggruppati in diversi package secondo dei criteri che verranno dopo definiti. Ulteriormente ai sorgenti relativi al progetto, in questa directory sarà contenuta un secondo package radice relativo al testing che verrà descritto nei prossimi paragrafi. Entrambi i package radice e tutto il loro contenuto verranno sottoposti a intensivo testing e versioning.
- **.gitignore:** file descrittore per l'applicativo git utilizzato per il versioning in team, esso conterrà una descrizione rigorosa di quali file non devono essere soggetti al versioning, motivo di ciò è per garantire una maggiore portabilità, modularità e leggerezza al progetto Sagrada in modalità versioned.
- **pom.xml:** file descrittore del progetto Sagrada utilizzando il framework Maven 3, esso conterrà, oltre ad una struttura descrittiva del progetto Sagrada in XML per garantire la portabilità tra i diversi ambienti Maven, la lista di dipendenze utilizzate e necessarie per la corretta build del progetto insieme al successo nell'esecuzione di ogni Maven test.
- **sonar-project.properties:** file descrittore della revisione automatizzata del progetto Sagrada attraverso il software SonarQube. Esso conterrà la locazione dei file .class contenente Java bytecode e la locazione dei sorgenti .java.

### 1.2.2 Organizzazione di massima del codice sorgente

All'interno della directory src che racchiude tutto il codice è necessario organizzare i file secondo una struttura ben precisa che evidenzia tutti i moduli operativi che compongono lo sviluppo in Java.

L'organizzazione del codice si attesta su una struttura a più livelli, implementata attraverso l'utilizzo dei costrutti di packaging tipici del linguaggio Java.

La struttura generale dei sorgenti può essere riassunta con questo primo schema:

```
src..... Directory che raccoglie tutti i sorgenti. Vedi punto precedente
|
|___ main/java.....Non contiene sorgenti, serve solo per raggruppare i package
|
|   |___ interfaces..... (M) package contenente interfacce base
|   |
|   |___ abstracts..... (M) package contenente classi astratte base
|   |
|   |___ network..... (M) package contenente la struttura delle comunicazioni di rete
|   |
|   |___ animations.... (M) package contenente le classi costituenti decorator per oggetti animabili
|   |
|   |___ ui.....(M) package contenente classi per la principale struttura d'interfaccia grafica.
|   |
|   |___ fx.....(M) package contenente classi costituenti oggetti grafici JavaFX
```

Per ogni MP (eventualmente, come sopra mostrato):

```
.....|
      |___ nome sotto package.... (SP) Sub package per migliore organizzazione di MP
```

**(MP)** = main package    **(SP)** = sub package

Si noti in particolare:

- Ogni main package fa riferimento a un preciso modulo del progetto ed ha lo scopo di fornire un contenitore comune a tutti i sorgenti sviluppati nel contesto di un preciso progetto.
- I nomi dei package sono costruiti da una radice identificante in modo chiaro il ruolo del package all'interno del progetto, se è eventualmente presente un sub package viene su di esso applicato lo stesso tipo di regola del main package. La creazione di un nuovo package deve essere valutata con attenzione e deve essere decisa di comune accordo tra gli sviluppatori. In generale vale la regola che ogni package raggruppa un preciso filone dello sviluppo Java e quindi è frutto di una precisa attività di analisi.
- L'alfabeto utilizzabile nella denominazione dei package è quello inglese di soli caratteri minuscoli

### 1.3 Convenzioni relative al linguaggio Java

Allo scopo di migliorare la comprensibilità e l'interscambio del codice sviluppato è opportuno adottare nella scrittura dei sorgenti alcune regole di stile.

#### 1. Nome delle classi: il nome delle classi deve essere il più possibile significativo e deve dare un'idea di massima della funzione svolta dalla classe stessa.

Attenersi pertanto alle seguenti regole:

- Evitare nomi troppo lunghi, senza significato o contenenti caratteri speciali che potrebbero dare problemi.
- Il nome della classe deve iniziare con una maiuscola così come ogni parola che compone il nome. Un esempio un nome corretto può essere **ObjectFieldListener** mentre indubbiamente la forma **objectfieldlistener** è meno leggibile e quindi preferibilmente da evitare.
- Il nome della classe deve possibilmente contenere anche un riferimento alla sua funzione all'interno del linguaggio: ad esempio, se una classe definisce un listener di un oggetto è opportuno che il suo nome finisca con la locuzione **Listener**; altri oggetti per cui può valere questa regola sono le Interface, gli Event, le Exception. Esempio: la classe che definisce una eccezione avrà un nome del tipo **PippoException**. Viene lasciato come discussione nel team la scelta dell'aggiunta del prefisso o postfisso **Abstract** nel caso di denominazione di classi astratte. Altri esempi: NetworkController, NetworkService, NetworkManager ecc...

#### 2. Nome dei metodi.

- Evitare nomi lunghi e con caratteri speciali.
- Il nome del metodo deve iniziare con una lettera minuscola. Se il nome è composto da più parole le parole intermedie devono però iniziare con una maiuscola. Ad esempio, un nome valido per un metodo può essere **printObjectList**.
- Seguire il più possibile lo standard dei JavaBean, in modo da facilitare l'eventuale utilizzo di una classe all'interno di sistemi introspettivi (si rimanda alla documentazione dei **Java Beans**):
  - I nomi dei metodi che assegnano il valore di una variabile di classe sono del tipo **set + Nome variabile**: ad esempio setCode è il metodo che assegna un valore all'attributo codice. Da notare che, se presente, la prima lettera delle variabili viene ignorata perché relativa al contesto della variabile (vedi sezione sulle variabili per maggiori dettagli).
  - I nomi dei metodi che leggono il valore di una variabile di classe sono del tipo **get + Nome variabile**: stesso principio per il caso dei metodi setter.
  - I nomi dei metodi che leggono il valore di una variabile logica sono del tipo **is + NomeVariabile**: ad esempio, il metodo che torna lo stato del flag

isAllocated sarà del tipo isAllocated(). I metodi di questo tipo devono sempre tornare un valore di tipo Boolean.

- Se il metodo prevede più combinazioni nei parametri di ingresso ripetere n volte la dichiarazione del metodo, ogni volta con il numero minimo di parametri di ingresso. Evitare la scrittura di metodi con molti parametri se si sa che gran parte di quei parametri si useranno solo in casi particolari: meglio scrivere una nuova dichiarazione del metodo da usarsi solo in quei casi.

### 3. Nome delle variabili e degli attributi.

- Al solito, evitare nomi troppo lunghi.
- Il nome deve iniziare con una lettera minuscola. Se il nome è composto da più parole le parole intermedie devono però iniziare con una maiuscola. È possibile utilizzare in certi casi solamente il carattere '\_'.
- Il nome assegnato alla variabile/attributo deve dare un'idea di dove sia dislocata la variabile/attributo stessa all'interno della classe. Quindi:
  - **Attributi di classe:** sono le variabili definite a livello di classe e che sono pertanto disponibili in ogni punto del sorgente. Il loro nome può avere una lettera iniziale per una maggiore localizzazione.
  - **Variabili locali:** sono le variabili definite all'interno di un metodo e valide solo all'interno del metodo in cui sono definite. Il loro nome può avere una lettera iniziale per una maggiore localizzazione.
  - **Attributi passati ad un metodo:** sono le variabili che definiscono i campi di input previsti da un metodo. Il linguaggio Java prevede che valgano solo all'interno del metodo in cui sono definite. Ad esempio:

```
public void printHash(String hash)
```

- **Costanti:** sono variabili dichiarate di tipo static e final. In questo caso il nome è tutto maiuscolo. Esempio:

```
public static final String FONT_LOCALE = "/fonts";
private static final String FONT_LOCALE = "/fonts";
```

È da considerare la stessa politica di scrittura in maiuscolo il nome delle costanti nelle enumerazioni.

- **Contatori:** le regole precedenti trovano eccezione nel caso di variabili usate per definire i contatori all'interno di costrutti di loop. In questo caso è conveniente utilizzare nomi brevi ma significativi, come **index**, **counter**, **key**...  
Nel caso di loop annidati utilizzare contatori diversi. Per esempio:



```

for (int counter = 0; counter < 10; counter ++ )
{
    for (int _counter = 10; _counter > 0; _counter --)
    {
        .....
    }
}

```

Si può eventualmente utilizzare nomi brevi di una lettera quali i, j, k... nel caso si venga alla decisione garantisca una maggiore leggibilità al codice oltre al fatto della non importanza della denominazione di un certo indice.

- Dichiarare preferibilmente variabili protette o private e definire gli opportuni metodi di set o di get se si vuole rendere pubblico l'accesso alle variabili da classi esterne. Nel codice scritto evitare se possibile l'accesso diretto alle variabili di una classe esterna, cioè assegnamenti del tipo `Classe.variabile = valore`, perché non compatibile con le specifiche JavaBeans (anche se sintatticamente corretto).

Ad esempio, il codice:

```

MyClass myClass = new myClass();
myClass.myAttribute = "Java";

```

è sintatticamente corretto ma non viene riconosciuto da un sistema di introspezione che si aspetta la presenza di un metodo `setMyAttribute(String attributeValue)` definito all'interno della classe `MyClass`. In questo caso, il codice corretto per l'assegnamento di una variabile diventa allora del tipo:

```

MyClass myClass = new MyClass();
myClass.setMyAttribute("Java");

```

#### 4. Scrittura e semantica dei sorgenti

- Evitare l'implementazione di metodi troppo lunghi, si cerca di spingere ad un'implementazione di progetto costituita da molte classi con poco codice, e di conseguenza metodi piccoli che implementano la minima funzionalità necessaria

- Non scrivere metodi con molti parametri, se la firma del metodo contiene più di 4/5 parametri è necessario valutare se è il caso di incapsulare questi parametri in un'apposita classe (DTO/Model)
- Ritornare oggetti strutturati ove necessario
- Utilizzare le parentesi graffe dove necessario il loro utilizzo deve essere affiancato durante l'apertura alla linea di codice corrente, esempio:

```

try {
    //codice
}
catch(MyException exc) {
    //altro codice
}

class MyClass {
    //codice
}

```

- Viene lasciato almeno uno spazio tra i vari metodi dichiarati e una linea vuota iniziale per ogni tipo di dichiarazione o espressione d'utilizzo, nessuno spazio tra gli attributi tranne uno spazio divisorio per gruppi di attributi diverso tipo. Esempio: attributi e valori costanti.
- Viene lasciato uno spazio bianco per ogni cambio di parametro ed espressione e sempre uno dopo una virgola ma non prima di essa.
- Nel caso dei costruttori : non verrà mai lasciato il costruttore di default, verrà sempre implementato un proprio costruttore, anche nel caso in cui funge da iniziatore di attributi con nessun parametro formale richiesto.
- Rendere quanto più possibile funzionale il codice attraverso l'utilizzo delle nuove funzionalità introdotte dal JDK 8
- Favorire l'utilizzo di Enumeration al posto di Boolean quando si hanno espressioni multi valore, esempio:

```
String getDate(Date date, Boolean beforeMidnight);
```

deve diventare:

```

enum TimeBound {
    MIDNIGHT,
    BEFORE_MIDNIGHT
}

String getDate(Date date, TimeBound timeBound);
stringDateFormat = getDate(myDate, TimeBound.BEFORE_MIDNIGHT);

```

- Condurre le espressioni all'interno di variabili booleane: cercare di lasciare le condizioni di valutazione booleane il più pulite e chiare possibili, esempio:

```
booleanCondition = evaluate();  
if (booleanCondition)  
  doSomething();
```

- Le righe del codice devono essere **sempre** indentate attraverso l'utilizzo di un tabulatore. Nonostante molti IDE utilizzino anche la politica di indentazione a 4 spazi, nel suddetto progetto verrà ammessa solamente l'indentazione attraverso il tabulatore.

## 5. Implementazione

- È sempre e comunque necessario ricordare di rispettare una certa complessità temporale, sebbene si possa utilizzare in pre RC durante il versioning e quindi temporaneamente una complessità temporale pari a  $O(N^2)$ , la complessità temporale a cui attenersi nella RC è pari a  $O(N \log N)$ .
- È sempre necessario un buon utilizzo di Java Collections che sfruttino inoltre il polimorfismo dovuto alla loro implementazione gerarchica, ovvero come esempio: se si volesse utilizzare un ArrayList come tipo di attributo o tipo di ritorno o qualsiasi altro valido utilizzo è necessario dichiarare questo attributo o variabile come List e non ArrayList, segue lo stesso ragionamento per tutti gli altri tipi di strutture dati.
- È necessario un buon utilizzo dei tipi generici in Java introdotti con il JDK 5 e inoltre il loro miglioramento con le wildcard introdotto con il JDK 7.
- Si richiede una forte modularizzazione e indipendenza tra le varie classi del progetto, questo necessario per avere un'ottima e semplicistica implementazione del progetto a moduli intercambiabili.
- È necessario l'utilizzo di vari design pattern per poter andare a compimento del punto precedentemente descritto.
- È **assolutamente vietato** il non rispetto delle caratteristiche della programmazione a oggetti riguardante la visibilità degli attributi di classe e riguardante i tre principi fondamentali: ereditarietà, polimorfismo e incapsulamento.

## 6. Gestione delle eccezioni

- È necessario dichiarare le specifiche eccezioni che un metodo può lanciare.

**Assolutamente vietato** gestire la sola eccezione generica `Exception`.

- Utilizzare il blocco `finally` per rilasciare le eventuali risorse acquisite
- È assolutamente vietato l'utilizzo di blocchi di `catch` senza alcun codice al loro interno
- Effettuare correttamente il `wrap` e `log` di un'eccezione, in modo da non perdere il suo `stack trace`.

Se questo non è possibile o va contro l'implementazione in via di scrittura l'eccezione dovrà essere rilanciata al chiamante.

- È necessario dichiarare un nuovo tipo d'eccezione quando il suo eventuale lancio è previsto dall'implementazione.

### 1.4 Documentazione dei sorgenti

Al fine di migliorare la comprensibilità dei sorgenti è utile inserire all'interno del codice degli opportuni commenti che facilitino la comprensione delle procedure.

Questo vale in particolar modo nel caso di procedure complesse o particolari.

La documentazione si sviluppa su due livelli distinti:

#### 1.4.1 Documentazione a livello di classe

Il file sorgente di ogni classe deve riportare in testata una serie di informazioni che consentano di identificare le funzioni svolte da una singola classe. La scelta di associare o meno una descrizione alla classe deve essere valutata caso e per caso e tenere conto sia dell'importanza della classe stessa sia della riusabilità del codice prodotto. La descrizione deve essere posta in testa al sorgente e racchiusa all'interno di commenti Java attraverso l'utilizzo dei costrutti previsti dal linguaggio. Le informazioni contenute nella testata devono essere le seguenti:

1. Data di creazione della classe
2. Autore
3. Descrizione di massima della funzione svolta dalla classe. Il livello di meticolosità della descrizione deve essere proporzionato alla effettiva importanza della classe all'interno del progetto (ad esempio, evitare descrizioni dettagliate per classi di servizio o di scarsa riusabilità).
4. Se necessario, una descrizione dei parametri di ingresso previsti dalla classe e del loro effetto sul comportamento della classe stessa.

5. Se necessario, un esempio di utilizzo della classe. È sufficiente riportare qualche riga di codice che rappresenti un tipico caso di utilizzo della classe, possibilmente senza entrare troppo nel dettaglio.
6. Una lista delle modifiche maggiori che sono state apportate alla classe, con particolare attenzione alle modifiche che hanno introdotto modifiche sostanziali. Questa lista può integrare le descrizioni inserite nel server di gestione dei sorgenti al momento del rilascio di una classe.

### 1.4.2 Documentazione a livello di metodo

All'interno di una classe può essere utile inserire dei commenti a livello di singolo metodo. Questo diventa necessario soprattutto quando il metodo svolge funzioni importanti oppure prevede un gran numero di parametri di difficile comprensione. La documentazione a livello di metodo si deve limitare a fornire informazioni che rendano semplice l'utilizzo del metodo senza dover analizzare il codice in esso contenuto: deve quindi definire con buon dettaglio cosa fa un metodo, quali sono le informazioni da fornirgli e qual è l'effetto dell'esecuzione di un metodo sul resto della classe. La documentazione a livello di metodo è utile che abbia un formato di tipo **javadoc** (vedi documentazione Java per i dettagli), in modo da facilitare una eventuale generazione automatica di documentazione operativa.

### 1.4.3 Altre forme di documentazione

Oltre ai casi visti in precedenza, ci possono essere altri casi particolari in cui può essere utile introdurre dei commenti ad un sorgente.

- È buona norma commentare i passaggi del codice che potrebbero essere di difficile interpretazione. Questo non vuol dire inserire commenti per ogni cosa che si fa ma solo per quei casi che potrebbero generare confusione ad un osservatore esterno.
- Segnalare sempre le parti di codice non completate o da sviluppare ulteriormente. Lo scopo è quello di inserire nel codice delle note che segnalino allo sviluppatore i punti da sviluppare ulteriormente e che consentano ad un osservatore esterno di capire se una funzione è completamente implementata o se manca qualcosa. Un metodo semplice può essere quello di inserire prima della parte incompleta un commento del tipo:

**//TODO:** descrizione delle parti mancanti previa  
implementazione

## **1.5 IDE, Testing, Repository e software ausiliari**

Al fine di migliorare il processo di sviluppo del progetto Sagrada è necessario aggiungere una serie di punti riguardanti gli strumenti per lo sviluppo.

### **1.5.1 IDE**

Per il progetto Sagrada verrà utilizzato l'IDE IntelliJ IDEA 2018 Community Edition. Per una più facile integrazione tra i vari host è vietato l'utilizzo di IDE alternativi.

### **1.5.2 Testing**

Nel progetto Sagrada il testing verrà effettuato attraverso il supporto di Maven 3. Esso è parte integrante del progetto ed è assolutamente necessario ad ogni nuova implementazione o aggiunta di classe o metodo o della loro relativa implementazione, aggiungere o cambiare le modalità e i processi di testing, in modo adeguato, descritti dalle classi di testing nel package radice main/test. Il superamento dei test garantirà una buona implementazione dell'astrazione precedentemente pensata. Non è possibile aggiungere alla repository comune sorgenti che non supereranno i test definiti. L'aggiornamento verrà rifiutato in modo automatico e notificato via e-mail.

### **1.5.3 Repository**

Il versioning e il repository della codebase del progetto Sagrada verrà effettuata attraverso l'utilizzo di git e messo disponibili in rete attraverso GitHub alla seguente repository privata.

È necessario che ad ogni cambiamento rilevante ed aggiunta venga aggiornata il repository comune.

La gestione dei vari comandi è sia utilizzabile in modalità integrata attraverso l'IDE sia via linea di comando.

### **1.5.4 Software ausiliari**

Per lo sviluppo del progetto verranno utilizzati due servizi e software ausiliari.

#### **1. SonarQube**

SonarQube verrà utilizzato attraverso un'integrazione nell'IDE utilizzato in collegamento su una sessione avviata localmente. Esso permetterà il miglioramento dell'intero insieme di sorgenti del progetto andando a identificare oltre ad eventuali falle presenti e che sono state o ignorate o sfuggite ad uno o più sviluppatori all'interno del team, anche i punti dove il sorgente permette un miglioramento a livello di complessità sia spaziale che temporale. I cambiamenti richiesti e suggeriti da SonarQube verranno resi visibili automaticamente ad ogni tentativo di commit, build e avvio del progetto facilitando le correzioni di volta in volta da lui suggerite ad ogni nuovo scanning effettuato.

#### **2. Travis CI**

Travis CI è un servizio remoto che permetterà il testing del progetto globalmente prima di far sì che un commit venga definitivamente alla codebase principale od altri branch. Se il testing fallirà il commit verrà rifiutato effettuando una rollback allo stato precedente della codebase e una notifica di errore verrà inviata via e-mail.

Questo permetterà la non presenza di codice non funzionante, supponendo che il test sia stato correttamente scritto, nella codebase principale ed altri relativi branch.

#### **3. Wireshark**

Wireshark è un software per il capturing dei pacchetti ed è stato necessario per verificare prima di tutto la corretta esecuzioni di una struttura a più protocolli multi flusso e senz'altro per andare a risolvere i problemi software debuggando per così dire la parte ingegnerizzata della rete.