INGEGNERIA INFORMATICA

ESAME DI

Ingegneria del Software

SIMULAZIONE CAMPIONATO DI UNA SOCIETÁ SPORTIVA TRAMITE UTILIZZO DI STRATEGY, OBSERVER E SINGLETON

Docente:
Enrico Vicario
Dipartimento di Ingegneria dell'Informazione

Studente:
Giovanni Stefanini
6182949

Indice

1. Introduzione

L'elaborato è la realizzazione di un applicativo, scritto in Java, che simula l'andamento del campionato di una squadra, per verificare l'efficacia di un allenatore.

1.1 Motivazioni e Contenuti

Una Società Sportiva è organizzata in Squadre e in Staff: le Squadre giocano un campionato suddiviso in Categorie (Juniores, Seniores). Lo Staff si divide in Allenatori e Dottori. Gli Allenatori preparano Allenamenti per una sola squadra. I Dottori sono assegnati a una sola squadra e osservano i giocatori di quella squadra. I giocatori e lo staff appartengono a una Squadra. Una società di Rugby ha un solo Presidente che decide lo Staff per ogni Squadra, prima dell'inizio del Campionato.
Ogni campionato è suddiviso per categorie. Ogni Squadra partecipa al Campionato della categoria corrispondente. Ogni campionato si risolve in modo diverso in base alla categoria scelta. La Squadra gioca una partita alla volta. Dopo ogni partita i giocatori possono subire un infortunio, che deve essere notificato al Dottore il quale dovrà segnare quanti infortuni ci sono stati nel campionato. Ogni partita può essere vinta, persa o pareggiata. In base all'esito dell'ultima partita l'allenatore deciderà che tipo di allenamento fare. .
Al termine del Campionato il Consiglio di ciascuna Categoria verifica la qualità degli allenamenti svolti dagli Allenatori guardando il numero di partite vinte, il numero di infortuni e aggiungendo un giudizio (negativo o positivo) sull'operato dell'allenatore, e produce un resoconto. Il resoconto viene poi letto dal Presidente. Il Presidente può così riconfermare lo Staff per il Campionato successivo o decidere di modificarlo.
La scelta di questo progetto è dettata da una motivazione personale, facendo parte di una società sportiva come allenatore di una squadra giovanile di rugby, e dalla necessità di pianificare il tipo di allenamento e la formazione della mia squadra.

1.2 Metodo

Per la realizzazione di tale applicazione è stato utilizzato il linguaggio di programmazione Java attraverso l'IDE Eclipse. .
Nella fase di analisi dei requisiti sono stati identificati i casi d'uso e rappresentati attraverso gli Use Case Diagrams. .
È stata inoltre definita e realizzata una logica di dominio in prospettiva di specifica/implementazione attraverso un diagramma UML. .
In questa fase sono stati disegnati alcuni Mockup di come dovesse essere l'interfaccia finale dell'applicativo utente. .

Nella struttura principale del programma sono stati adottati alcuni pattern comportamentali come il pattern Observer tre la classe giocatore e la classe dottore, il pattern Strategy per eseguire il campionato in base alla categoria scelta. Inoltre è stato usato il pattern creazionale Singleton per la classe Presidente. L'utente dovrà così inserire da tastiera il nome del Presidente, dell'Allenatore, del Dottore, della Squadra, il numero dei giocatori e in fine scegliere una Categoria. Nel progetto sono anche gestiti in modo casuale alcuni comportamenti non predicibili, infatti nella realtà gli allenatori dopo ogni partita dovrebbero inserire il risultato e di conseguenza fare un allenamento specifico, i giocatori dovrebbero vedere quando realmente si infortunano per avvisare il dottore e il consiglio dovrebbe dare il giudizio in base al comportamento dell'allenatore e in base al materiale umano, ma nel programma tutto ciò viene simulato. .

2. Requisiti

2.1 Casi d'Uso

Dopo aver definito il problema in questione sono stati individuati gli attori in gioco e i vari casi d'uso.

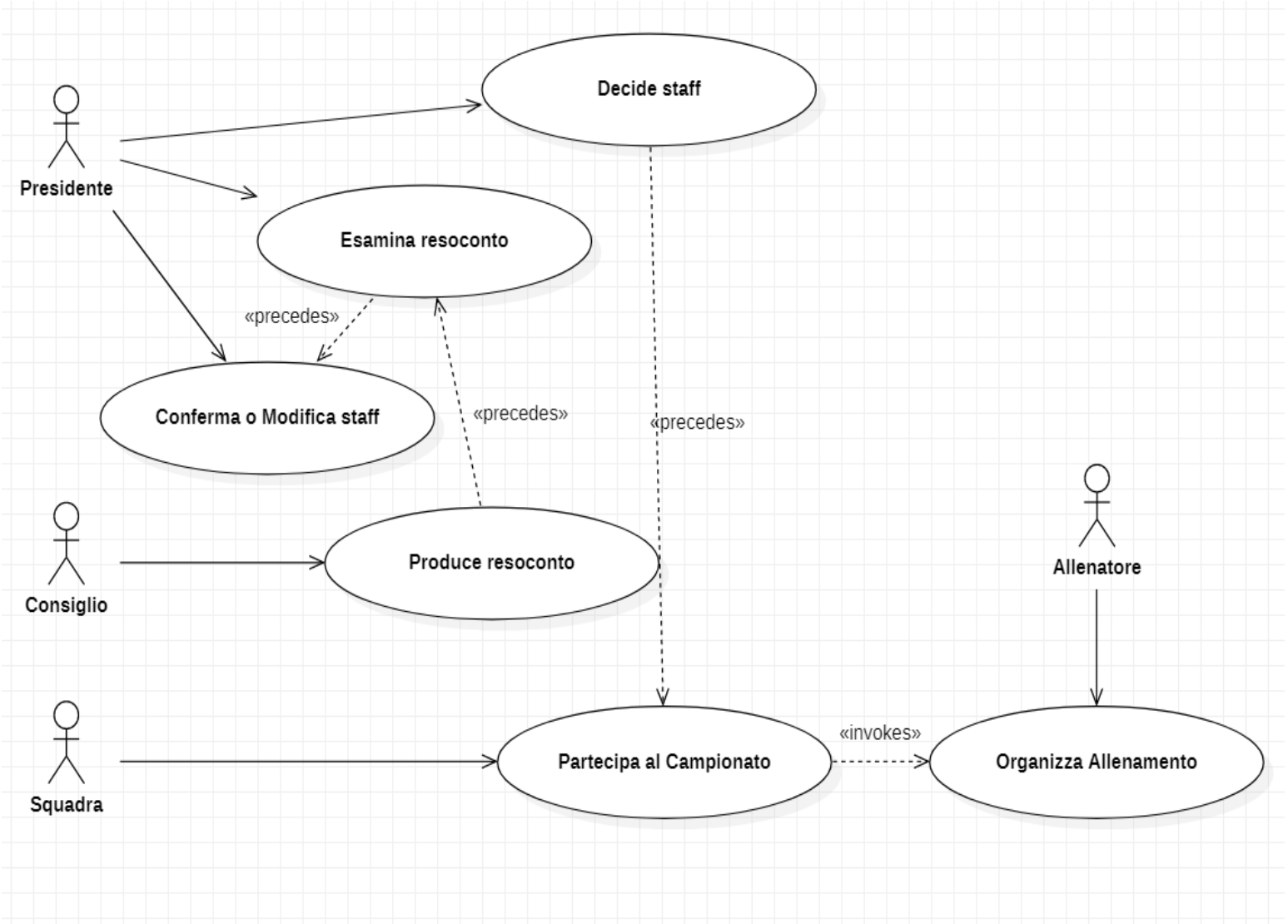

Figura 1. Use Case Diagram

ATTORE PRIMARIO: .

è colui che inizia l'azione, ed è rappresentato dal *presidente* deve poter decidere lo saff all'inizio del campionato. Inoltre deve esaminare il resoconto prodotto dal consiglio alla fine del campionato per poter infine decidere se confermare o modificare lo staff. .

ATTORI SECONDARI: .
il *consiglio* produce il resoconto alla fine del campionato. .
La *squadra* partecipa al campionato dopo che il presidente ha deciso lo staff che la seguirà.
L'allenatore in base alla partita giocata dalla squadra organizzerà l'allenamento.

Espandiamo con use case templates il caso d'uso DECIDE STAFF. .
USE CASE TEMPLATES: .
Name Use Case: Decide staff. .
Level: User goal. .
Descrizione: il presidente dovrà decidere lo staff per una squadra, cioè dottore e allenatore, inserendo il nome di quest'ultimi. .
Actors: Presidente. .
Pre-Conditions: il presidente si dovrà essere identificato fornendo il nome. .
Post-Conditions: avendo creato lo staff, si dovrebbe poter creare la squadra per partecipare al campionato. .
Normal flow: .
0. Il caso inizia quando il cliente apre la pagina e immette il nome del presidente. .
1. Il presidente decide lo staff. .
2. Il presidente immette il nome dell'allenatore. .
3. Il presidente immette il nome del dottore. .
4. Il cliente decide il nome della squadra e il numero di giocatori. .

2.2 Diagramma UML

Qui di seguito la relazione del diagramma UML che descrive la logica di dominio in prospettiva di implementazione.

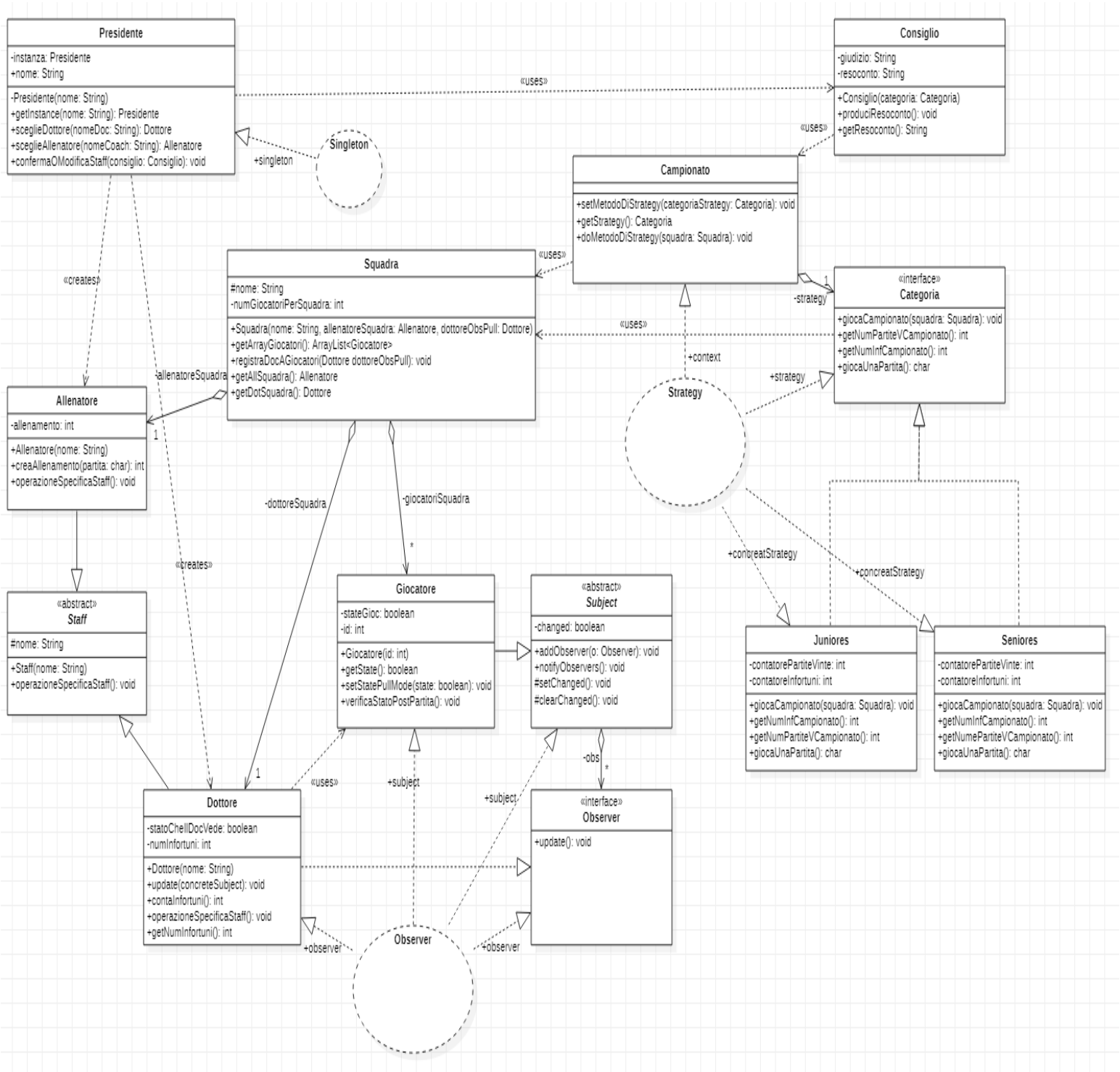

Figura 2. Class Diagram

2.3 Mock-Ups

Sono stati realizzati alcuni Mockups dell'interfaccia utente finale tramite l'utilizzo di Balsamiq.

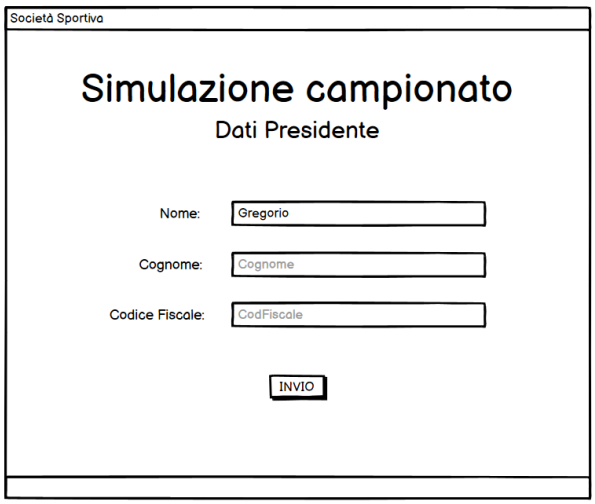

Figura 3. Mock-Ups Interfaccia utente

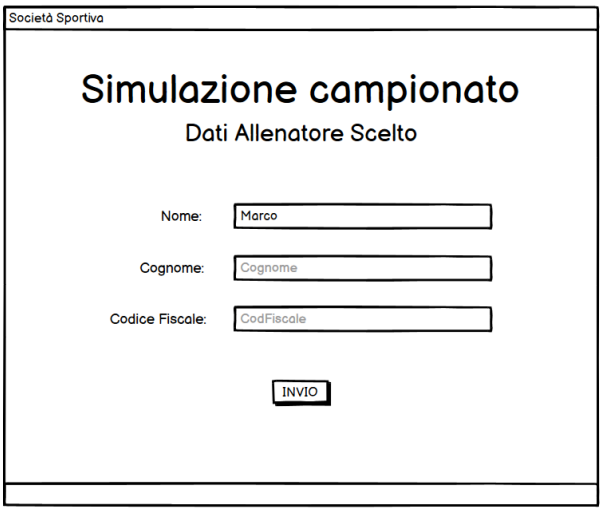

Figura 4. Mock-Ups interfaccia utente

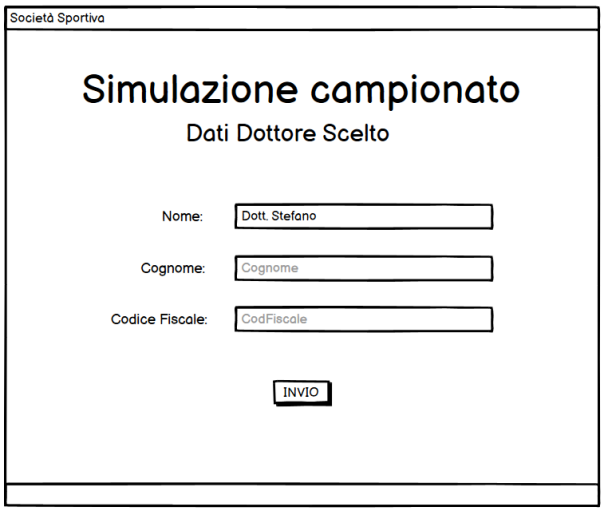

Figura 5. Mock-Ups Interfaccia utente

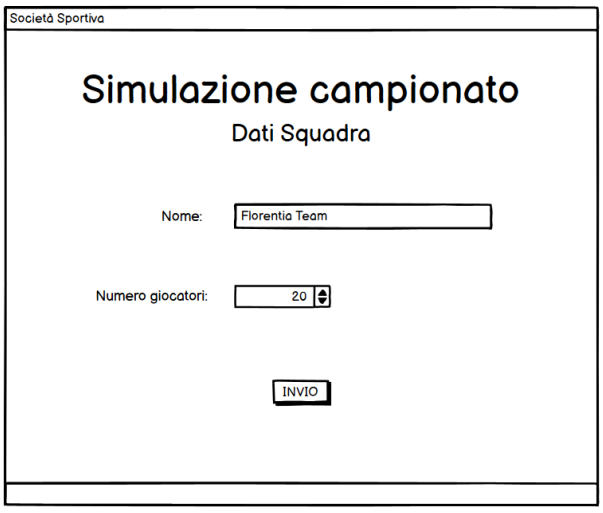

Figura 6. Mock-Ups interfaccia utente

Figura 7. Mock-Ups Interfaccia utente

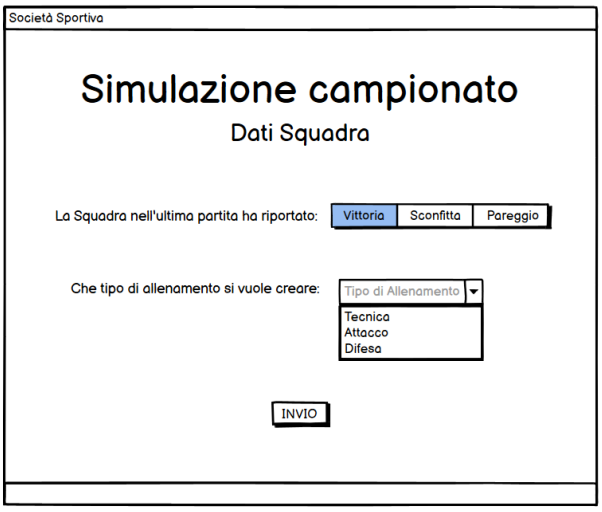

Figura 8. Mock-Ups interfaccia utente

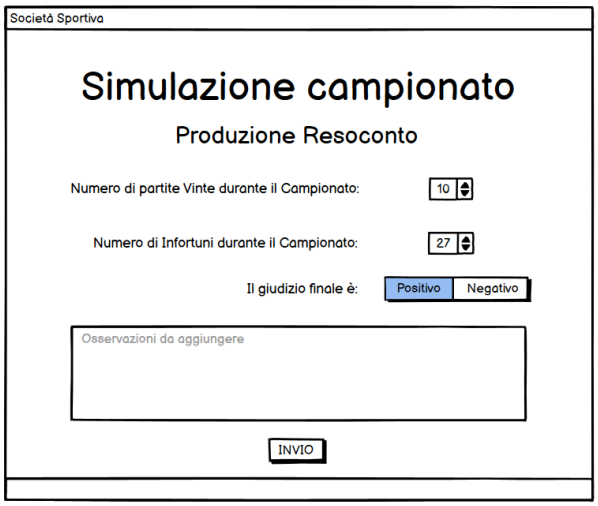

Figura 9. Mock-Ups Interfaccia utente

Figura 10. Mock-Ups interfaccia utente

Figura 11. Mock-Ups Interfaccia utente

3. Implementazione

3.1 Design Pattern

Per l'implementazione sono stati usati i seguenti design pattern *Strategy*, *Singleton* e *Observer* che vengono richiamati di seguito, indicando come sono stati implementati.

3.1.1 Strategy

Si tratta di un pattern comportamentale che Consente la definizione di una famiglia d'algoritmi, incapsulandone ognuno e rendendoli intercambiabili fra di loro. Questo permette di modificare gli algoritmi in modo indipendente dai Client che fanno uso di essi. Il pattern Strategy suggerisce l'incapsulamento della logica di ogni particolare algoritmo, in apposite classi (ConcreteStrategy) che implementano l'interfaccia che consente agli oggetti Context di interagire con loro. Questa interfaccia deve fornire un accesso efficiente ai dati del Context, richiesti da ogni ConcreteStrategy, e viceversa.
Struttura:

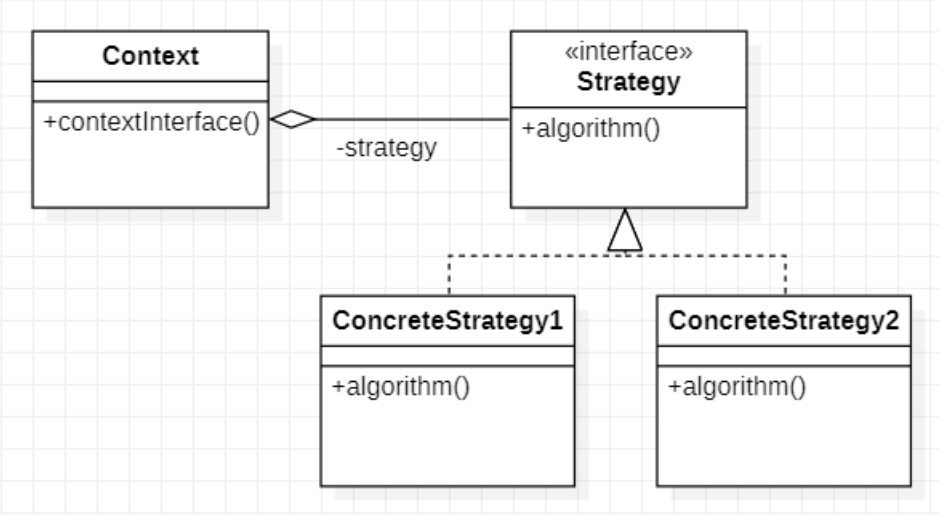

Figura 12. Class diagram Strategy

Partecipanti:
· *Strategy*: dichiara un'interfaccia comune per tutti gli algoritmi supportati. Il Context utilizza questa interfaccia per invocare gli algoritmi definiti in ogni ConcreteStrategy.
· *ConcreteStrategy*: implementano gli algoritmi usando l'interfaccia Strategy.
· *Context*: viene configurato con un oggetto ConcreteStrategy e mantiene un riferimento verso esso.
Può specificare una interfaccia che consenta a Strategy di accedere ai propri dati.

Implementazione:
Nel progetto abbiamo implementato questo design pattern al fine di poter far decidere al cliente quale tipo di campionato giocare in base alla categoria selezionata, potendo così adattare il campionato.

Le classi dei pattern sopra citati sono così corrisposte nell'implementazione:
-Strategy: Categoria;
-ConcreteStrategy1: Juniores;
-ConcreteStrategy2: Seniores;
-Context: Campionato;

```
package esameSWE_GiovanniStefanini;

public class Campionato {

    private Categoria strategy;

    public void setMetodoDiStrategy(Categoria categoriaStrategy) {
        this.strategy = categoriaStrategy;
    }
    public Categoria getStrategy() {
        return strategy;
    }
    public void doMetodoDiStrategy(Squadra squadra){
        strategy.giocaCampionato(squadra);
    }
}
```

Figura 13. Implementazione classe Campionato

```
package esameSWE_GiovanniStefanini;

public interface Categoria {
    //è lo strategy
    public void giocaCampionato(Squadra squadra);

    public int getNumPartiteVCampionato();

    public int getNumInfCampionato();

    public char giocaUnaPartita();

}
```

Figura 14. Implementazione classe Categoria

3.1.2 Singleton

Il Singleton è un pattern creazionale che ha il fine di restringere l'istanziazione di una classe ad un unico oggetto, ed assicurare così che qualora ne venga richiesto un altro, venga sempre ritornato un riferimento all'istanza iniziale.

Struttura:

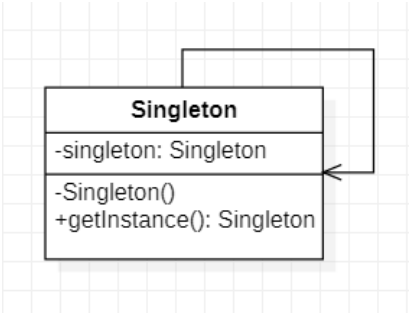

Figura 15. Class Diagram Singleton

Partecipanti:

· *Singleton*: è la classe di cui vogliamo avere una unica istanza.

Implementazione:

È stato implementato il pattern Singleton, sia per fini didattici sia per avere la garanzia che nell'ambiente di test esista uno ed uno solo presidente. Inoltre, è stato utilizzando un costruttore privato (Item 2) avendo così la garanzia che la classe possa essere istanziata solo da un metodo interno a essa, chiamata nell'implementazione *getInstance()*; tale costruttore infatti non potrà essere ereditato e dunque usato da eventuali sottoclassi, né potrà essere invocato dall'esterno. .

In fine l'utilizzo del pattern Singleton sopra descritto è stato fatto al fine di evitare la creazione di oggetti duplicati non necessari (Item 4). .

Le classi dei pattern sopra citati sono così corrisposte nell'implementazione:

-*Singleton: Presidente*; .

```java
package esameSWE_GiovanniStefanini;

public class Presidente {

    private static Presidente instance = null;
    protected String name;

    private Presidente(String name) {
        this.name = name;
        System.out.println("Presidente creato: " + name);
    }

    public static final Presidente getInstance(String name) {
        if (instance == null)
            instance = new Presidente(name);
        return instance;
    }
```

Figura 16. Snippets classe Presidente

3.1.3 Observer

L'Observer è un pattern comportamentale che Consente la definizione di associazioni di dipendenza di molti oggetti verso di uno, in modo che se quest'ultimo cambia il suo stato, tutti gli altri sono notificati e aggiornati automaticamente. Permette quindi di evitare ambiguità. Può essere implementato in forma Push in cui la notifica dal soggetto porta il cambiamento di "stato" (*notify(state)*) e dunque la modifica di alcuni parametri, per distinguere tra più soggetti, e in modo Pull in cui il soggetto notifica che è avvenuta una modifica, e l'Observer fa la call-back (chiamata di ritorno), se necessario, in accordo con la sua politica. Un comune side-effect del partizionamento è la necessità di mantenere consistenza e coerenza tra oggetti correlati, senza bisogno di ricorrere a classi strettamente accoppiate per preservare la riusabilità.
Struttura:

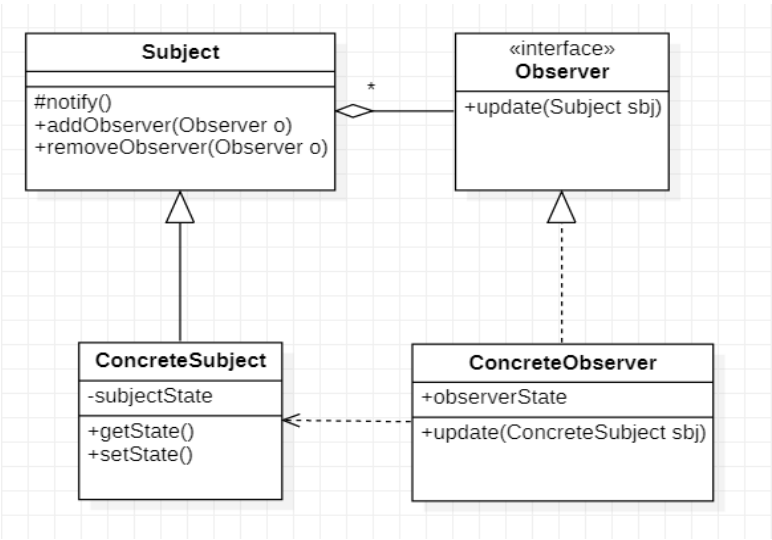

Figura 17. Class Diagram Observer

Partecipanti:
· *Subject*: conosce i suoi observers, i quali possono esserci in numero illimitato. Fornisce un'interfaccia per il collegamento e lo scollegamento degli oggetti Observer (*addObserver()* e *removeObserver()*). Fornisce operazioni per la notifica agli Observer.
· *Observer*: definisce un'interfaccia per la notifica di eventi agli oggetti interessati in un Subject, e un'interfaccia per la notifica a tutti i vari Observer. Il Subject ha un array di riferimenti a questi Observer.
· *ConcreteSubject*: memorizza lo stato di interesse per oggetti ConcreteObserver ed invoca le operazioni di notifica ereditate dal Subject quando devono essere informati i ConcreteObserver.
· *ConcreteObserver*: memorizza lo stato che dovrebbe rimanere coerente e sincrono a quello del Subject e implementa l'interfaccia di *update()* Observer per mantenere il suo stato in linea con quello del Subject.

Implementazione:
Solitamente l'Observer si usa quando una modifica ad un oggetto richiede il cambiamento di altri oggetti e non si sa quanti oggetti hanno bisogno di essere cambiati. In altre parole, non si desidera che questi oggetti siano strettamente accoppiati.
Nell'Implementazione del progetto l'Observer è stato implementato in modo Pull e usato per permette a un Dottore di vedere lo stato di un Giocatore che poteva cambiare da "in forma" a "infortunato" dopo ogni partita. Così facendo tutte le volte che un Giocatore si Infortuna il Dottore avrà modo di curarlo e di segnare il numero di infortuni importante ai fini della simulazione. Può permettere, in un futuro, di aumentare il numero di Dottori e quindi di poter aggiungere osservatori senza modificare il soggetto (Giocatore) o altri osservatori. .

Le classi dei pattern sopra citati sono così corrisposte nell'implementazione:
-Subject: Subject;
-Observer: Observer;
-ConcreteSubject: Giocatore;
-ConcreteObserver: Dottore;

```
public abstract class Subject {

    private boolean changed = false;
    private Vector<Observer> obs;

    public Subject() {
        obs = new Vector<>();
    }

    public synchronized void addObserver(Observer o) {
        if (o == null)
            throw new NullPointerException();
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }

    public void notifyObservers() {
        Object[] arrLocal;

        synchronized (this) {
            if (!changed)
                return;
            arrLocal = obs.toArray();
            clearChanged();
        }

        for (int i = arrLocal.length-1; i>=0; i--)
            ((Observer)arrLocal[i]).update(this);
    }

    protected synchronized void setChanged() {
        changed = true;
    }

    protected synchronized void clearChanged() {
        changed = false;
    }
    public synchronized boolean hasChanged() {
        return changed;
    }

}
```

Figura 18. Implementazione della classe Subject

```java
public class Giocatore extends Subject{
    //è il concreteSubject
    public static final boolean inFORMA = true;
    public static final boolean INFORTUNATO = false;

    private boolean stateGioc;
    private int id;

    public Giocatore(int i) {
        this.id=i;
        stateGioc = inFORMA;
    }

    public boolean getState() {
        return stateGioc;
    }

    public void setStatePullMode(boolean state) {
        this.stateGioc=state;
        setChanged();
        notifyObservers();
    }
```

Figura 19. Snippets della classe Giocatore

```java
public interface Observer {

    public void update(Subject concreteSubject);

}
```

Figura 20. Implementazione della classe Observer

```java
public class Dottore extends Staff implements Observer  {

    private boolean statoCheIlDocVede;
    private int numInfortuni = 0;

    public static final boolean INFORTUNATO = false;
    public static final boolean inFORMA = true;

    public Dottore (String nome) {
        super(nome);
    }


    public void update(Subject concreteSubject) {
        this.statoCheIlDocVede = ((Giocatore)concreteSubject).getState();

        if(statoCheIlDocVede == INFORTUNATO) {
            System.out.println("Un giocatore va da "+this.nome+" perchè da in FORMA è diventato INFORTUNATO");
            contaInfortuni();
            operazioneSpecificaStaff();
        }
    }
```

Figura 21. Snippets della classe Dottore

3.2 Classi ed interfacce

Le altre classi che non implementano un pattern contenute nel package `esameSWE_GiovanniStefanini` sono: .

-1 Presidente
-2 Consiglio
-3 Squadra
-4 Seniores
-5 Juniores
-6 Staff
-7 Allenatore
-8 EsameSWE_GiovanniStefanini

3.2.1 Presidente

La classe Presidente viene implementata realizzando il pattern Singleton perché vogliamo essere sicuri che ci sia un'unica istanza di essa. .

```
public class Presidente {

    private static Presidente instanza = null;
    protected String nome;

    private Presidente(String nome) {
        this.nome = nome;
        System.out.println("Presidente creato: " + nome);
    }

    public static final Presidente getInstance(String name) {
        if (instanza == null)
            instanza = new Presidente(name);
        return instanza;
    }

    public Dottore sceglieDottore(String nomeDoc) {
         Dottore dottoreScelto = new Dottore(nomeDoc);
         return dottoreScelto;
    }

    public Allenatore sceglieAllenatore(String nomeCoach) {
         Allenatore allenatoreScelto = new Allenatore(nomeCoach);
         return allenatoreScelto;
    }

    public char confermaOModificaStaff(Consiglio consiglio) {
        char decisione;
        String resocontoConsiglio = consiglio.getResoconto();
        if(resocontoConsiglio == "Confermato") {
            System.out.println("Lo Staff può essere riconfermato per l'anno prossimo");
            decisione = 'c';
        } else {
            System.out.println("Lo Staff deve essere modificato per l'anno prossimo");
            decisione = 'm';
        }
        return decisione;
    }

}
```

Figura 22. Implementazione classe Presidente

La responsabilità del Presidente è quella di decidere lo Staff (Dottore e Allenatore) all'inizio della simulazione del campionato che realizza con i metodi *sceglieDottore()* e *sceglieAllenatore(),* e di confermare o modificare lo Staff alla fine del campionato stesso, dopo che il Consiglio ha prodotto il resoconto con il metodo *confermaOModificaStaff().* .

3.2.2 Consiglio

La classe Consiglio ha il compito alla fine del campionato di produrre il resoconto tramite il metodo *produciResoconto()* in base al numero di partite vinte e di infortuni durante il campionato, in più aggiunge un giudizio positivo o negativo. In base alla combinazione di queste variabili il resoconto potrà avere come risultato una String con valore "Confermato" o "Negato". .

```
public class Consiglio {

    private String giudizio;
    private String resoconto;

    public void produciResoconto(Campionato campionato) {
        Categoria categoria = campionato.getStrategy();
        int contatorePartiteVinte = categoria.getNumPartiteVCampionato();
        int contatoreInfortuni = categoria.getNumInfCampionato();

        double numCasuale = (int)(Math.random()*10);
        if(numCasuale<7) {
            giudizio = "POSITIVO";
        } else giudizio = "NEGATIVO";

        System.out.println("Giudizio è " +giudizio);

        if(contatorePartiteVinte>6 && contatoreInfortuni<70 && giudizio == "POSITIVO") {
            resoconto = "Confermato";
        } else if(contatorePartiteVinte>10 && contatoreInfortuni<50 && giudizio == "NEGATIVO") {
            resoconto = "Confermato";
        } else resoconto = "Negato";

        System.out.println("Resoconto è " +resoconto);
    }

    public String getResoconto() {
        return resoconto;
    }

}
```

Figura 23. Implementazione classe Consiglio

3.2.3 Squadra

La Squadra è una classe che contiene al suo interno l'allenatore scelto dal Presidente, il Dottore scelto dal Presidente, il numero di giocatori da cui è composta e il nome che gli viene assegnato. Nel momento della sua creazione viene creato una ArrayList di Giocatori per creare i nuovi giocatori che faranno parte della squadra. Inoltre questa classe tramite il metodo *registraDocAGiocatori()* fa in modo che il

dottore della squadra diventi l'Observer di tutti i giocatori che la compongono. La squadra così incapsula informazioni che poi serviranno per eseguire il campionato.

```
public class Squadra {

    protected String nome;
    private int numGiocatoriPerSquadra;
    private ArrayList<Giocatore> giocatoriSquadra;
    private Allenatore allenatoreSquadra;
    private Dottore dottoreSquadra;

    public Squadra(String nome, Allenatore allenatoreSquadra, Dottore dottoreObsPull, int numGiocatoriSq) {
        this.nome = nome;
        giocatoriSquadra = new ArrayList<Giocatore>();
        this.allenatoreSquadra = allenatoreSquadra;
        this.dottoreSquadra = dottoreObsPull;
        this.numGiocatoriPerSquadra = numGiocatoriSq;
        for(int i=0; i < numGiocatoriPerSquadra; i++) {
            giocatoriSquadra.add(new Giocatore(i));
        }
        registraDocAGiocatori(dottoreObsPull);
    }

    public void registraDocAGiocatori(Dottore dottoreObsPull) {
         for(int i=0; i < giocatoriSquadra.size(); i++) {
             giocatoriSquadra.get(i).addObserver(dottoreObsPull);
         }
    }
```

Figura 24. Snippets classe Squadra

3.2.4 Seniores

La classe Seniores fa parte della collaboration di Strategy, infatti implementa l'interfaccia Categoria andando così a rappresentare il concreateStrategy. Nello specifico, tra i metodi di cui fa l'override da Categoria quello importante è il metodo *giocaCampionato()* così da implementare lo strategy che verrà realizzato nel caso in cui sia scelta questa classe. Questo metodo ha bisogno che gli venga passata una Squadra e, utilizzando le informazioni in essa, effettua la simulazione del campionato. A tal fine chiama in modo iterativo il metodo *giocaUnaPartita()* che permette di simulare l'andamento di una partita in modo casuale differente da Juniores.

```java
public class Seniores implements Categoria {

    private int contatorePartiteVinte;
    private int contatoreInfortuni;

    @Override
    public void giocaCampionato(Squadra squadra) {
        System.out.println("INIZIO Campionato Seniores si gioca 25 partite");
        char risUltimaPartita;
        for(int i=0; i<25; i++) {
            System.out.println("Partita n° " +i);
            risUltimaPartita = giocaUnaPartita();
            if (risUltimaPartita == 'V') {
                contatorePartiteVinte++;
            }
            System.out.println("Risultato ultima partita: " + risUltimaPartita);
            for(int j=0; j < squadra.getArrayGiocatori().size(); j++) {
                squadra.getArrayGiocatori().get(j).verificaStatoPostPartita();
            }
            squadra.getAllSquadra().creaAllenamento(risUltimaPartita);
            System.out.println("------------------------");
        }
        System.out.println("Numero partite vinte nel campionato: " + contatorePartiteVinte);
        System.out.println("Numero infortuni: " + squadra.getDotSquadra().getNumInfortuni());

        contatoreInfortuni = squadra.getDotSquadra().getNumInfortuni();
    }

    @Override
    public char giocaUnaPartita() {
         char risPartita;
         double numCasuale = (int)(Math.random()*10); //crea un numero casuale da 1 a 10
         if(numCasuale<3) {
             risPartita = 'S';
         } else if(numCasuale<8) {
             risPartita = 'V';
         } else risPartita = 'P';
        return risPartita;
    }
```

Figura 25. Snippets classe Categoria

3.2.5 Juniores

La classe Juniores fa parte della collaboration di Strategy, infatti implementa l'interfaccia Categoria andando così a rappresentare il concreateStrategy. Nello specifico, tra i metodi di cui fa l'override da Categoria quello importante è il metodo *giocaCampionato()* così da implementare lo strategy che verrà realizzato nel caso in cui sia scelta questa classe. Questo metodo ha bisogno che gli venga passata una Squadra e, utilizzando le informazioni in essa, effettua la simulazione del campionato. A tal fine chiama in modo iterativo il metodo *giocaUnaPartita()* che permette di simulare l'andamento di una partita in modo casuale differente da Seniores.

```java
public class Juniores implements Categoria {
    private int contatorePartiteVinte;
    private int contatoreInfortuni;

    @Override
    public void giocaCampionato(Squadra squadra) {
        System.out.println("INIZIO Campionato Juniores si giocano 15 partite");
        char risUltimaPartita;
        for(int i=0; i<15; i++) {
            System.out.println("Partita n° " +i);
            risUltimaPartita = giocaUnaPartita();
            if (risUltimaPartita == 'V') {
                contatorePartiteVinte++;
            }
            System.out.println("Risultato ultima partita: " + risUltimaPartita);

            for(int j=0; j < squadra.getArrayGiocatori().size(); j++) {
                squadra.getArrayGiocatori().get(j).verificaStatoPostPartita();
            }
            squadra.getAllSquadra().creaAllenamento(risUltimaPartita);
            System.out.println("------------------------");
        }
        System.out.println("Numero partite vinte nel campionato: " + contatorePartiteVinte);
        System.out.println("Numero infortuni: " + squadra.getDotSquadra().getNumInfortuni());

        contatoreInfortuni = squadra.getDotSquadra().getNumInfortuni();
    }

    @Override
    public char giocaUnaPartita() {
         char risPartita;
         double numCasuale = (int)(Math.random()*10); //crea un numero casuale da 1 a 10
         if(numCasuale<3) {
             risPartita = 'P';
         } else if(numCasuale<6) {
             risPartita = 'S';
         } else risPartita = 'V';
        return risPartita;
    }
```

Figura 26. Snippets classe Categoria

3.2.6 Staff

Staff è una classe astratta da cui ereditano Allenatore e Dottore. Definisce una default implementation del metodo *operazioneSpecificaStaff()* anche se ne viene fatto l'override nelle classi derivate. Serve nel caso non volessimo distinguere tra Allenatore e Dottore.

```java
public abstract class Staff {

    protected String nome;

    public Staff(String nome) {
        this.nome = nome;
        System.out.println("Nome Staff ingaggiato: " + nome);
    }

    public void operazioneSpecificaStaff() {
        System.out.println("lo staff si consulta");
    }

    public String getName() {
        return nome;
    }
}
```

Figura 27. Implementazione classe Staff

3.2.7 Allenatore

La classe Allenatore estende la classe Staff. .
Ha il compito di creare un allenamento in base al risultato dell'ultima partita giocata. A tal fine implementa il metodo *creaAllenamento()* a cui passiamo il risultato dell'ultima partita. Inoltre questo metodo chiama il metodo *operazioneSpecificaStaff()* di cui ne viene fatto l'Override da Staff. .

```java
public class Allenatore extends Staff {

    private int allenamento;

    public Allenatore(String nome) {
        super(nome);
    }

    public int creaAllenamento(char partita) {
        operazioneSpecificaStaff();

        if(partita =='S') {
            allenamento = 1;
            System.out.println("Allenamento sulla Difesa");
        } else if(partita == 'V') {
            allenamento = 2;
            System.out.println("Allenamento sull'Attacco");
        } else {
            allenamento = 3;
            System.out.println("Allenamento sulla Tecnica individuale");
        }
        return allenamento;
    }

    @Override
    public void operazioneSpecificaStaff() {
        System.out.println("Allenatore pensa al tipo di allenamento");
    }
}
```

Figura 28. Implementazione classe Allenatore

3.2.8 EsameSWE_GiovanniStefanini

Rappresenta la classe che contiene il *main* del progetto. Nel main prendiamo da tastiera i valori che ci interessano per poter creare le classi che poi andranno a effettuare la simulazione del campionato di cui possiamo scegliere la categoria a cui la squadra parteciperà.

```
public class EsameSWE_GiovanniStefanini {

    public static void main(String[] args) {
        String nomePresidente;
        String nomeAllenatore;
        String nomeDottore;
        String nomeSquadra;
        int numGiocatoriSq;
        int categoriaScelta;
        System.out.println("Inizio");
        Scanner input = new Scanner(System.in);
        System.out.println("Inserisci nome Presidente: ");
        nomePresidente = input.nextLine();
        System.out.println("Inserisci nome Allenatore: ");
        nomeAllenatore = input.nextLine();
        System.out.println("Inserisci nome Dottore: ");
        nomeDottore = input.nextLine();
        System.out.println("Inserisci nome Squadra: ");
        nomeSquadra = input.nextLine();
        System.out.println("Inserisci numero giocatori per Squadra: ");
        numGiocatoriSq = input.nextInt();
        System.out.println("Inserisci categoria a cui si vuole partecipare 1 = juniores e qualsiasi Intero = seniores: ");
        categoriaScelta = input.nextInt();
        Presidente presidente = Presidente.getInstance(nomePresidente);
        Dottore dottoreQualsiasi = presidente.sceglieDottore(nomeDottore);
        Allenatore allenatoreQualsiasi = presidente.sceglieAllenatore(nomeAllenatore);
        Squadra squadraQualsiasi = new Squadra(nomeSquadra, allenatoreQualsiasi, dottoreQualsiasi, numGiocatoriSq);
        Campionato campionatoQualsiasi = new Campionato();
        if(categoriaScelta == 1) {
            Juniores categoriaQualsiasi = new Juniores();
            campionatoQualsiasi.setMetodoDiStrategy(categoriaQualsiasi);
            campionatoQualsiasi.doMetodoDiStrategy(squadraQualsiasi);
        } else {
            Seniores categoriaQualsiasi = new Seniores();
            campionatoQualsiasi.setMetodoDiStrategy(categoriaQualsiasi);
            campionatoQualsiasi.doMetodoDiStrategy(squadraQualsiasi);
        }
        Consiglio consiglioQualsiasi = new Consiglio();
        consiglioQualsiasi.produciResoconto(campionatoQualsiasi);
        presidente.confermaOModificaStaff(consiglioQualsiasi);
        System.out.println("Fine");
    }
}
```

Figura 29. Implementazione main

4. Unit Testing

Quando si parla di *Unit Testing* si intende la verifica del comportamento di una classe tramite test di unità, cioè di singole porzioni di codice nello specifico dei metodi. Una volta individuate le varie sezioni di codice si potrà procedere con i test che vengono detti test case. .
Nel progetto è stato utilizzato il framework *JUnit 5*. .
In JUnit i test-case sono dei metodi anteposti dalle annotazioni *@Test* ma JUnit mette a disposizione anche altre annotazioni come per esempio *@BeforeEach, @AfterEach,* etc.
Per ogni classe che è stata testata del progetto sono state create le rispettive classi di test contenenti i test case relativi ai metodi della classe da testare.
In ogni metodo di test vengono verificate delle asserzioni (*asserts*) elementari attraverso vari metodi offerti da JUnit stesso: *assertEquals, assertTrue, assertFalse, assertNull* etc. .
Le classi di test nel progetto sono: .
-1 PresidenteTest
-2 GiocatoreTest
-3 CampionatoTest
-4 SquadraTest
Sono state scelte queste quattro classi di test perché sono quelle che permettono di coprire più casi d'uso e permettono di testare più classi del codice.

4.1 PresidenteTest

Testando la classe Presidente si testa la Collaboration del Pattern Singleton. Per testare questa classe sono stati testati i metodi *getInstance(), sceglieDottore(), sceglieAllenatore()* e *confermaOModificaStaff().* Nello specifico il test *getInstanceTest* server per verificare il corretto funzionamento del Singleton, mentre gli altri metodi servono per verificare il corretto funzionamento del comportamento della classe. In particolare eseguendo *confermaOModificaStaffTest* verifichiamo anche il comportamento della classe Consiglio chiamando il suo metodo *produciResoconto().*

```java
class PresidenteTest {
    private Presidente pres;
    private String nomeDot;
    private String nomeAll;
    private Consiglio cons;
    @BeforeEach
    public void initialize() {
        pres = Presidente.getInstance("Presidente");
        nomeDot = "Dot";
        nomeAll = "All";
        cons = new Consiglio();
    }
    @Test
    public void getInstanceTest() {
        Presidente presidenteTest = Presidente.getInstance("Presidente Test");
        assertSame(pres, presidenteTest);
    }
    @Test
    public void sceglieDottoreTest() {
        Dottore dot;
        dot = pres.sceglieDottore(nomeDot);
        assertEquals(nomeDot, dot.getName());
    }
    @Test
    public void sceglieAllenatoreTest() {
        Allenatore all;
        all = pres.sceglieAllenatore(nomeAll);
        assertEquals(nomeAll, all.getName());
    }
    @Test
    public void confermaOModificaStaffTest() {
        Squadra sq = new Squadra("Sq N", new Allenatore("All"), new Dottore("Dot"), 10);
        Campionato camp = new Campionato();
        Juniores cat = new Juniores();
        camp.setMetodoDiStrategy(cat);
        camp.doMetodoDiStrategy(sq);
        cons.produciResoconto(camp);
        pres.confermaOModificaStaff(cons);
        assertNotNull(pres.confermaOModificaStaff(cons));
    }
}
```

Figura 30. Implementazione PresidenteTest

4.2 GiocatoreTest

Viene testata la classe Giocatore al fine di testare la Collaboration del pattern di Observer. Infatti testando il metodo *setStatePullMode()* con i test *setStatePullModeTest1*, *setStatePullModeTest2* e *setStatePullModeTest3* vengono testate anche i metodi delle classi Subject, Observer e Dottore. Nello specifico, di Subject, viene testato: *setChanged()* e *notifyObservers();* di Observer, viene testato: *update();* di Dottore, viene testato: *update(), contaInfortuni()* e *operazioneSpecificaStaff().*

Inoltre viene anche testato il metodo *verificaStatoPostPartita()* di questa classe.

```
class GiocatoreTest {
    public static final boolean inFORMA = true;
    public static final boolean INFORTUNATO = false;
    private Giocatore gioc;
    private Dottore dot;
    private int id;
    @BeforeEach
    public void initialize() {
        id = 1;
        gioc = new Giocatore(id);
        dot = new Dottore("Dot");
        gioc.addObserver(dot);
    }
    @Test
    public void setStatePullModeTest1() {
        gioc.setStatePullMode(false);
        assertFalse(gioc.getState());
    }
    @Test
    public void setStatePullModeTest2() {
        gioc.setStatePullMode(false);
        assertFalse(dot.getStatoCheIlDocVede());
    }
    @Test
    public void setStatePullModeTest3() {
        gioc.setStatePullMode(false);
        assertEquals(gioc.getState(), dot.getStatoCheIlDocVede());
    }
    @Test
    public void verificaStatoPostPartitaTest() {
        gioc.verificaStatoPostPartita();
        assertEquals(gioc.getState(), dot.getStatoCheIlDocVede());
    }
    @AfterEach
    public void setStateGioc() {
        gioc.setStatePullMode(true);
    }
}
```

Figura 31. Implementazione GiocatoreTest

4.3 CampionatoTest

Nell'elaborato è stata testata la classe Campionato in quanto con essa riusciamo a testare la Collaboration che realizza il pattern Strategy. Nello specifico, con i test *doMetodoDiStrategyTestS* e *doMetodoDiStrategyTestJ* si testano anche le classi Categoria, Juniores e Seniores, poiché testando il metodo *doMetodoDiStrategy()* si nota che esso invoca i metodi delle classi suddette nello specifico *giocaCampionato().* Inoltre viene testato il metodo *setMetodoDiStrategy()* per vedere se viene impostata

correttamente la strategia selezionata. .

```
public class CampionatoTest {
    private Campionato camp;
    private Dottore doc;
    private Allenatore all;
    private Squadra sq;
    private int numGiocSq;
    @BeforeEach
    public void initialize() {
        camp = new Campionato();
        doc = new Dottore("Doc. D");
        all = new Allenatore ("All. A");
        numGiocSq = 10;
        sq = new Squadra("Sq. T", all, doc, numGiocSq);
    }
    @Test
    public void setMetodoDiStrategyTest() {
        //Verifica che la strategia sia correttemente scelta
        Seniores strategy = new Seniores();
        camp.setMetodoDiStrategy(strategy);
        assertEquals(camp.getStrategy(), strategy);
    }
    @Test
    public void doMetodoDiStrategTestS() {
        //Testa un campionato Seniores
        Campionato campSeniores = new Campionato();
        Seniores strategyS = new Seniores();
        campSeniores.setMetodoDiStrategy(strategyS);
        campSeniores.doMetodoDiStrategy(sq);
        assertNotNull(campSeniores);
    }
    @Test
    public void doMetodoDiStrategyTestJ() {
        //Testa un campionato Juniores
        Campionato campJuniores = new Campionato();
        Seniores strategyJ = new Seniores();
        campJuniores.setMetodoDiStrategy(strategyJ);
        campJuniores.doMetodoDiStrategy(sq);
        assertNotNull(campJuniores);
    }
}
```

Figura 32. Implementazione CampionatoTest

4.4 SquadraTest

Per testare la classe Squadra è stato verificato il comportamento del metodo *registraDocAGiocatori()* che viene invocato alla creazione della Squadra. Questo metodo deve abilitare il servizio offerto dal pattern Observer registrando il Dottore scelto ai giocatori della Squadra. .
Quindi al fine di verificarne l'effettiva funzionalità è stato verificato che un

cambiamento dello stato dei giocatori viene notificato al Dottore, tramite l'uso di tre test *registraDocAGiocatoriTest1*, *registraDocAGiocatoriTest2* e *registraDocAGiocatoriTest3*.

```
class SquadraTest {
    private Squadra sq;
    private ArrayList<Giocatore> giocatoriSquadra;
    private int numGSq;
    private Dottore dotSq;
    @BeforeEach
    public void initialize() {
        dotSq = new Dottore("Dot");
        numGSq = 10;
        sq = new Squadra("nomeSq", new Allenatore("All"), dotSq, numGSq);
        giocatoriSquadra = sq.getArrayGiocatori();
    }
    @Test
    public void registraDocAGiocatoriTest1() {
        for(int i=0; i < giocatoriSquadra.size(); i++) {
            giocatoriSquadra.get(i).setStatePullMode(false);
            assertFalse(giocatoriSquadra.get(i).getState());
        }
    }
    @Test
    public void registraDocAGiocatoriTest2() {
        for(int i=0; i < giocatoriSquadra.size(); i++) {
            giocatoriSquadra.get(i).setStatePullMode(false);
            assertFalse(dotSq.getStatoCheIlDocVede());
        }
    }
    @Test
    public void registraDocAGiocatoriTest3() {
        for(int i=0; i < giocatoriSquadra.size(); i++) {
            giocatoriSquadra.get(i).setStatePullMode(false);
            assertEquals(dotSq.getStatoCheIlDocVede(), giocatoriSquadra.get(i).getState());
        }
    }
}
```

Figura 33. Implementazione SquadraTest