



**UNIVERSITÀ
DI TRENTO**

Dipartimento di Ingegneria e Scienza dell'Informazione

Progetto:

Animati



Titolo del documento:

Architettura

Gruppo:

T51

Indice

Diagramma delle Classi 3

OCL ?

Scopo del documento

[...]

Diagramma delle classi

Nel presente capitolo vengono presentate le classi previste nell'ambito del progetto Animati. Vengono riportate di seguito le classi individuate a partire dai diagrammi di contesto e dei componenti.

Classi enumerative di supporto

Unità

La classe **Unità** è una classe di supporto utilizzata nella classe Info, che sta ad indicare con i suoi attributi la durata di un'attività.

Ruolo

La classe **Ruolo** è una classe di supporto utilizzata nella classe Utente, che sta ad indicare con i suoi attributi il ruolo assunto da uno specifico utente.

Formato

La classe **Formato** è una classe di supporto utilizzata ogniqualvolta si deve indicare il formato di un file da esportare. In questo caso nella classe ListaAttività, quando viene esportata una lista col metodo `esporta(formato : Formato)`.

TipoDado

La classe **TipoDado** è una classe di supporto utilizzata nella classe Dado, che sta ad indicare con i suoi attributi il tipo di faccia utilizzata dallo strumento dado.

MetodoDivisione

La classe **MetodoDivisione** è una classe di supporto utilizzata nella classe CreazioneSquadre, che sta ad indicare con i suoi attributi la metodologia di divisione scelta dall'utente per l'estrazione delle squadre.

Stato

La classe **Stato** è una classe di supporto utilizzata nelle classi Cronometro e Timer, che serve per descrivere lo stato in cui si trovano gli stessi.

Classi di supporto

Data

La classe **Data** è una classe di supporto che con i suoi attributi giorno, mese, anno, orario sta ad indicare un preciso momento e che grazie al suo attributo `now()` restituisce i valori di questi attributi.

URL

La classe **URL** è una classe di supporto che con i suoi attributi protocollo e percorso va ad indicare un'immagine o un suono, a seconda dell'uso che si fa della classe.

Per esempio, nella classe Suono, l'url utilizzato per l'attributo sorgente rappresenta un suono. Al contrario, nelle classi Faccia e Utente, l'attributo immagine rappresenta, come indicato dal nome, un'immagine.

Time

La classe **Time** è una classe di supporto che con i suoi attributi, va a rappresentare un tempo con la precisione massima nell'ordine dei centesimi di secondo.

Colore

La classe **Colore** è una classe di supporto che con i suoi attributi, va a rappresentare un colore espresso tramite codice RGB, uno spazio di colore che riproduce i colori visibili all'uomo tramite la mescolanza additiva dei tre colori di base: rosso, verde e blu.

Info, Filtro ed Etichetta

La classe **Etichetta** è una classe di supporto che con i suoi attributi, va a rappresentare nome, descrizione e categoria di un'etichetta che può essere assegnata ad un'attività. Viene usata nella classe Info.

La classe **Info** è una classe di supporto che con i suoi attributi va a definire tutte le informazioni riguardanti un'attività.

La classe **Filtro** è una classe di supporto ed è collegata tramite una generalizzazione alla classe Info. Viene utilizzata per contenere le informazioni secondo le quali le attività devono essere filtrate.

Dado e Faccia

La classe **Faccia** è una classe di supporto alla classe Dado, e presenta tutti gli attributi necessari a definire qual è il tipo di una faccia del dado e cosa vi è rappresentato. La classe **Dado** è una classe il quale compito è quello di fornire gli attributi e i metodi necessari all'utilizzo dello strumento dado.

Una volta determinati i parametri definiti dagli attributi, grazie ai metodi presenti, il metodo estrai() è quello che fa funzionare lo strumento.

Cronometro

La classe **Cronometro** è una classe il quale compito è quello di fornire gli attributi e i metodi necessari all'utilizzo dello strumento cronometro.

Il tempo viene rappresentato grazie alla classe di supporto Time.

Timer e Suono

La classe **Suono** è una classe il quale compito è quello di fornire gli attributi e i metodi necessari all'utilizzo dello strumento fischiello, nonché di essere una classe di supporto alla classe Timer.

La classe **Suono**, grazie ai suoi attributi e metodi riesce a riprodurre un suono in base a come viene gestito l'attributo booleano inRiproduzione.

La classe **Timer** è una classe il quale compito è quello di fornire gli attributi e i metodi necessari all'utilizzo dello strumento timer.

I metodi stop() e start() di Timer hanno una funzione diversa rispetto a quelli di Suono, in quanto si occupano di fermare e avviare il timer e non di riprodurre o meno il suono.

CreazioneSquadre

La classe **CreazioneSquadre** è una classe il quale compito è quello di fornire gli attributi e i metodi necessari all'utilizzo dello strumento creazione squadre.

Gli attributi presenti indicano i valori dei parametri come anche se quei parametri sono stati impostati, nel caso

degli attributi booleani *Set*.

Grazie ai metodi presenti viene poi fatta l'estrazione delle squadre, secondo la metodologia scelta dall'utente.

SegnaPunti

La classe **SegnaPunti** è una classe il quale compito è quello di fornire gli attributi e i metodi necessari all'utilizzo dello strumento segna punti.

L'attributo *contatori* rappresenta i contatori delle varie squadre, che vengono incrementati e/o decrementati grazie ai metodi presenti.

Utente

La classe **Utente** è una classe che rappresenta colui che utilizza l'applicazione. Ci sono quindi attributi che rappresentano i dati identificativi di quell'utente, come anche il ruolo e lo stato, che può essere offline o online. Il metodo *login()* crea un'istanza di Autenticazione

Un utente può promuovere gli altri utenti, ma l'attributo *promossoDa*, serve nel caso un utente voglia declassare un altro utente che ha il ruolo di amministratore. In tal caso, l'utente deve essere quello che lo ha promosso a tale.

Un utente può creare una o più attività e/o liste di attività, rappresentate rispettivamente dalle classi *Attività* e *ListaAttività*.

Un utente può effettuare una o più segnalazioni e/o valutazioni, rappresentate rispettivamente dalle classi *Segnalazione* e *Valutazione*.

Autenticazione

La classe **Autenticazione** è una classe che rappresenta il processo di login di un utente.

Quando viene chiamato il metodo *login()* della classe *Utente*, viene creata un'istanza di Autenticazione e viene chiamata *richiestaAutorizzativa()*.

Se il codice è valido viene chiamato il metodo *richiestaToken()* e successivamente *dettagliAccount()* che mette i risultati negli attributi *id* e *mail* dell'utente.

Viene chiesto a MongoDB il ruolo dell'utente e la sua foto profilo che vengono a loro volta assegnati agli attributi *ruolo* e *immagine* dell'utente.

Infine avviene un aggiornamento dei dati locali.

Segnalazione

La classe **Segnalazione** è una classe che rappresenta la segnalazione fatta da un utente ad un'attività. Più segnalazioni possono riferirsi ad una stessa attività. Ogni segnalazione è stata effettuata da un solo utente.

Gli attributi rappresentano le informazioni relative alla segnalazione, ovvero descrizione, utente da cui è stata fatta e attività alla quale si riferisce.

Un utente può effettuare una segnalazione grazie al metodo presente.

Se un utente non esiste più le segnalazioni effettuate dallo stesso rimangono.

Valutazione

La classe **Valutazione** è una classe che rappresenta la valutazione fatta da un utente ad un'attività. Più valutazioni possono riferirsi ad una stessa attività. Un utente può dare una sola valutazione ad un'attività.

Gli attributi rappresentano le informazioni relative alla valutazione, ovvero il voto espresso con un numero intero, l'attività a cui si riferisce e l'utente da cui è stata fatta.

*Un utente può effettuare una segnalazione grazie al metodo presente.
Se un utente non esiste più le valutazioni effettuate dallo stesso rimangono.*

GestoreDatiOffline

*La classe **GestoreDatiOffline** è una classe che rappresenta tutte le operazioni che vengono fatte sui dati presenti localmente, ovvero che non necessitano che l'utente sia online.*

Attività

*La classe **Attività** è una classe che rappresenta tutto ciò riguardante un'attività.
Viene aiutata dalla classe di supporto Info, che contiene gran parte delle informazioni dell'attività stessa. A un'attività si possono riferire delle segnalazioni e/o valutazioni. Il metodo mostraSegnalazioni() mostra tutte le segnalazioni associate a quella specifica attività.
Il metodo divisioneSquadre, reindirizza l'utente alla schermata di creazione squadre con i parametri già riempiti per rispettare i vincoli di quella specifica attività.
Un'attività è contenuta nel catalogo e può essere contenuta in una lista. Più liste possono contenere la stessa attività, e un'attività può essere contenuta più volte nella stessa lista.
Un'attività viene creata da un solo utente.*

Catalogo

*La classe **Catalogo** è una classe che rappresenta il catalogo di attività.
Tra gli attributi c'è un'istanza di Filtro. Grazie al metodo filtra(filtro : Filtro) si va a riempire l'attributo lista con il catalogo filtrato.
Il catalogo può contenere delle attività.
Il metodo mostraAttivitàSegnolate(richiedente : Utente) mostra la lista di attività segnalate e per ogni attività quante sono le segnalazioni.*

ListaAttività

*La classe **ListaAttività** è una classe che rappresenta le liste di attività create dagli utenti.
Un utente può creare più liste di attività. Ne ha almeno una in quanto ogni utente ha la lista "Preferiti". Ogni lista ha un solo utente, non esistono liste condivise.
Tra gli attributi della lista sono presenti le informazioni che la identificano e con i metodi forniti si può crearla, rimuoverla, mostrarla ed esportarla, oltre ad aggiungervi o rimuovere attività.*

MongoDB

*La classe **MongoDB** è una classe che rappresenta in che modo il sistema si interfaccia con MongoDB.
L'unico attributo isConnected indica se si è connessi o meno al DBMS.
I metodi presenti rappresentano tutti i modi in cui il sistema interagisce con MongoDB.*

Codice in Object Constraint Language

In questo capitolo è descritta in modo formale la logica prevista nell'ambito di alcune operazioni di alcune classi. Tale logica viene descritta in Object Constraint Language (OCL) (<https://www.omg.org/spec/OCL/2.4/PDF>) perché tali concetti non sono esprimibili in nessun altro modo formale nel contesto di UML.

Cronometro

Metodo	Precondizioni	Postcondizioni
start()	stato deve essere in "reset" o in "pause"	stato assume valore "run"
pause()	stato deve essere in "run"	stato assume valore "pause"
stop()	stato deve essere in "pause"	<ul style="list-style-type: none"> stato assume valore "reset" parziali è una lista vuota
parziale()	stato deve essere in "run"	<ul style="list-style-type: none"> stato rimane al valore "run" aggiunto tempo a lista parziali

```
context Cronometro::start()
pre: (self.stato = "reset") OR (self.stato = "pause")
post: self.stato = "run"
```

```
context Cronometro::pause()
pre: self.stato = "run"
post: self.stato = "pause"
```

```
context Cronometro::stop()
pre: self.stato = "pause"
post: (self.stato = "reset") AND (self.parziali->isEmpty())
```

```
context Cronometro::parziale()
pre: self.stato = "run"
post: (self.stato = "run") AND (self.parziali = self.parziali@pre->append(self.tempo))
```

Segna-Punti

Invarianti:

contatori : Tuple{nome : String, punteggio : int}[0..N]

- contatori contiene al più 99 elementi
- il campo nome di ogni elemento di contatori deve avere tra 0 (escluso) e 99 (incluso) caratteri
- il campo punteggio di ogni elemento di contatori deve essere compreso tra -500 e 500


```
context Segna-Punti inv :
self.contatori->size() <= 99
```

```
context Segna-Punti inv :
self.contatori->forall(c : Tuple{nome : String, punteggio : int} | 0 < c.nome.size() AND
c.nome.size <= 99)
```

```
context Segna-Punti inv :
self.contatori->forall(c : Tuple{nome : String, punteggio : int} | -500 <= c.punteggio AND
c.punteggio <= 99)
```

Metodo	Precondizioni	Postcondizioni
aggiungiContatore(indice : int)	<ul style="list-style-type: none"> Il nome non deve essere vuoto e non può eccedere i 99 caratteri il numero di contatori può essere al massimo 98 	la sequenza di contatori deve avere come ultimo elemento un contatore con il nome scelto e punteggio pari a 0
incrementa(indice : int)	il contatore all'indice scelto deve avere punteggio minore di 500	il punteggio del contatore alla posizione scelta viene incrementato di 1
decrementa(indice : int)	il contatore all'indice scelto deve avere punteggio maggiore di -500	il punteggio del contatore alla posizione scelta viene decrementato di 1

```
context Segna-Punti::aggiungiContatore(indice : int)
pre: (0 < nome.size() AND nome.size() <= 99) AND (self.contatori->size() < 99)
post: self.contatori = self.contatori@pre->append(Tuple{nome : String = nome, punteggio : int = 0})
```

```
context Segna-Punti::incrementa(indice : int)
pre: self.contatori->at(indice) < 500
post: (self.contatori->at(indice)).punteggio = (self.contatori@pre->at(indice)).punteggio + 1
```

```
context Segna-Punti::decrementa(indice : int)
pre: self.contatori->at(indice) > -500
post: (self.contatori->at(indice)).punteggio = (self.contatori@pre->at(indice)).punteggio - 1
```

Timer

Invarianti:

suono : Suono

- il suono non può essere in riproduzione se il timer non è scaduto

```
context Timer inv :
self.suono.inRiproduzione implies self.stato = "run" AND self.tempo.ore = 0 AND
self.tempo.minuti = 0 AND self.tempo.secondi = 0 AND self.tempo.centhesimi = 0
```

Metodo	Precondizioni	Postcondizioni
start()	stato deve essere in "reset" o in "pause"	stato assume valore "run"
pause()	stato deve essere in "run"	stato assume valore "pause"
stop()	stato deve essere in "pause"	stato assume valore "reset"
imposta(ore : int, minuti : int, secondi : int)	il tempo fornito deve essere valido	tempo assume il valore del tempo fornito
spegni()	self.suono.inRiproduzione deve essere true	self.suono.inRiproduzione deve essere false e stato dev'essere reset

```
context Timer::start()
pre: (self.stato = "reset") OR (self.stato = "pause")
post: self.stato = "run"
```

```
context Timer::pause()
pre: self.stato = "run"
post: self.stato = "pause"
```

```
context Timer::stop()
pre: self.stato = "pause"
post: self.stato = "reset"
```

```
context Timer::imposta(ore : int, minuti : int, secondi : int)
pre: 0<=ore AND 0<=minuti AND minuti<60 AND 0<=secondi AND secondi<60
post: self.tempo = tempo
```

```
context Timer::spegni()
pre: self.suono.inRiproduzione
post: (NOT self.suono.inRiproduzione) AND stato = reset
```

Dado

Invarianti:

dimensione : int

- La dimensione deve essere compresa tra 0 e 9.999 inclusi

campione : Faccia[0..N]

- La dimensione del campione deve essere al più 9.999 incluso
- ogni elemento del campione deve avere tipo corrispondente al tipo del dado

estremoInferiore : int

- L'estremo inferiore deve essere compreso tra 0 e 9.999 inclusi

passo : int

- Il passo deve essere compreso tra 1 e 9.999 inclusi

estrazioni : int

- Il numero di estrazioni effettuate deve essere non negativo

```
context Dado inv :
0<=self.dimensione AND self.dimensione<=9.999 AND self.campione->size()<=9.999 AND
self.campione.forALL(f : Faccia | f.tipo = self.tipo)
```

```
0<=self.estremoInferiore AND self.estremoInferiore<=9.999 AND 1<=self.passo AND
self.passo<=9.999
AND 0<=self.estrazioni
```

Metodo	Precondizioni	Postcondizioni
scegliModalità(reimmissione : bool)	non deve essere stata effettuata ancora alcuna estrazione	l'attributo reimmissione viene impostato al valore selezionato
scegliTipo(tipo : tipoDado)	<ul style="list-style-type: none"> non deve essere stata effettuata ancora alcuna estrazione il campione dev'essere vuoto 	l'attributo tipo viene impostato al valore selezionato
scegliDimensione(dim : int)	<ul style="list-style-type: none"> non può essere stata effettuata ancora alcuna estrazione il valore fornito dev'essere una dimensione valida il campione dev'essere vuoto 	l'attributo dimensione viene impostato al valore selezionato
scegliEstremoPasso(estremo : int, passo : int)	<ul style="list-style-type: none"> non deve essere stata effettuata ancora alcuna estrazione 	<ul style="list-style-type: none"> gli attributi estremoInferiore e passo vengono impostati ai valori selezionati

	<ul style="list-style-type: none"> • i valori forniti devono essere validi • il tipo del dado dev'essere "numeri" • il campione dev'essere vuoto 	<ul style="list-style-type: none"> • il campione viene riempito con un numero di valori pari a dimensione che partono da estremoInferiore con scarto di passo tra due elementi consecutivi
aggiungiElemento(elemento : Faccia)	<ul style="list-style-type: none"> • non deve essere stata effettuata ancora alcuna estrazione • l'elemento fornito dev'essere del tipo del dado • il tipo del dado dev'essere diverso da "numeri" • la dimensione del campione dev'essere inferiore alla dimensione impostata 	Il campione deve avere come ultimo elemento l'elemento fornito
estrai() : Faccia	il campione non dev'essere vuoto	<ul style="list-style-type: none"> • il risultato è un elemento presente nel campione prima della chiamata al metodo • estrazioni viene incrementato di 1 • se reimmissione è false, un'occorrenza del risultato viene rimossa dal campione

```
context Dado::scegliModalità(reimmissione : bool)
pre: self.estrazioni = 0
post: self.reimmissione = reimmissione
```

```
context Dado::scegliTipo(tipo : tipoDado)
pre: self.estrazioni = 0 AND self.campione->isEmpty()
post: self.tipo = tipo
```

```
context Dado::scegliDimensione(dim : int)
pre: self.estrazioni = 0 AND 0<dim AND dim<=9.999 AND self.campione->isEmpty()
post: self.dimensione = dim
```

```
context Dado::scegliEstremoPasso(estremo : int, passo : int)
pre: self.estrazioni = 0 AND self.campione->isEmpty() AND 0<estremo AND estremo<=9.999 AND 1<passo AND passo<=9.99
post: self.estremoInferiore = estremo AND self.passo = passo AND self.campione->size() =
```

```
dimensione AND self.campione->first().numero = estremoInferiore AND self.campione-
>forall(f:Faccia | self.campione->count(f)=1 AND let i=self.campione->indexOf(f) in if i>1 then
f.numero=passo+self.campione->at(i-1) endif)
```

```
context Dado::aggiungiElemento(elemento : Faccia)
pre: self.estrazioni = 0 AND elemento.tipo = self.tipo AND self.tipo <> "numeri" AND
self.campione->size() < dimensione
post: self.campione = self.campione@pre->append(elemento)
```

```
context Dado::estrai() : Faccia
pre: self.campione->notEmpty()
post: self.campione@pre->includes(result) AND self.estrazioni = self.estrazione@pre+1 if (NOT
reimmissione) then (self.campione->size() = self.campione@pre->size() AND self.campione-
>forall(f:Faccia | let c=self.campione@pre->count(f) in if f<>result then self.campione-
>count(f)=c else self.campione->count(f)=c-1 endif)) endif
```

Suono

Invarianti:

dimensione : int

- la dimensione in byte del suono non deve superare i 100MB

```
context Suono inv:
self.dimensione <= 100.000.000
```

Metodo	Precondizioni	Postcondizioni
start()		il suono è in riproduzione
stop()		il suono non è in riproduzione
scegliSuono(sorgente : URL)		la sorgente del suono è quella scelta

```
context Suono::start()
pre:
post: self.inRiproduzione = true
```

```
context Suono::stop()
pre:
post: self.inRiproduzione = false
```

```
context Fischisuono::scegliSuono(sorgente : URL)
post: self.sorgente = sorgente
```

Creazione Squadre

Invarianti:

```
context Creazione Squadre inv :
numeroSquadre <= 99
numeroComponenti <= 99
numeroPartecipanti <= 9801
```

Metodo	Precondizioni	Postcondizioni
inserisciNumSquadre(numero : int)	numeroSquadre deve essere minore di 99	informazioni viene incrementato di 1
inserisciNumComponenti(numero : int)	numeroComponenti deve essere minore di 99	informazioni viene incrementato di 1
inserisciNumPartecipanti(numero : int)	numeroPartecipanti deve essere minore di 9801	informazioni viene incrementato di 1
scegliMetodo(metodo : Enum)		l'attributo metodo viene impostato con quello scelto
inserisciNome(nome : String)	la lunghezza dei nomi non deve eccedere i 99 caratteri	l'attributo nomi viene impostato con i nomi scelti
estrai()	<ul style="list-style-type: none"> informazioni deve essere uguale a 2 o 3 nel caso i tre valori: numeroSquadre, numeroPartecipanti e numeroComponenti non siano compatibili, verranno solamente considerati numeroPartecipanti e numeroSquadre. 	<ul style="list-style-type: none"> se metodo assume il valore "Random" l'ordine delle squadre assegnate sarà completamente casuale se metodo assume il valore "Round robin" l'assegnamento delle squadre sarà sequenziale se il metodo assume il valore "Fill first" l'assegnamento avverrà per completamento delle squadre, ovvero riempiendo i posti di ogni squadra prima di procedere con l'assegnamento per la prossima se il metodo assume il valore "Balanced" tutte le squadre dovranno avere lo stesso numero di partecipanti prima di

procedere con gli assegnamenti

- qualsiasi sia il metodo scelto, ogni volta che un partecipante viene assegnato ad una squadra il contatore di quella squadra viene incrementato di 1

Utente

ruolo : Enum

Invarianti:

- ruolo assume i valori "Amministratore" e "Base"

```
context Utente inv :
(ruolo = "Amministratore") OR (ruolo = "Base")
```

Metodo	Precondizioni	Postcondizioni
cambiaRuolo(utente, promotore, nuovoRuolo)	<ul style="list-style-type: none"> • un amministratore può essere declassato a utente comune unicamente dall'amministratore che lo ha promosso • se l'attributo ruolo dell'utente ha il valore "Amministratore", il suo ruolo può essere cambiato solo se promotore è diverso dall'attributo promossoDa dell'utente • il promotore deve avere ruolo "Amministratore" 	l'attributo ruolo dell'utente assume il valore di nuovoRuolo

Autenticazione

Metodo	Precondizioni	Postcondizioni
verificaOAuth()		
generaToken()		
ottieniUtente() : Utente		
logout()		

// TODO

Catalogo

Metodo	Precondizioni	Postcondizioni
aggiornaCatalogo() : Attività[0...N]		l'attributo ultimoAggiornamento assume il valore della data corrente
filtra(cerca : String, etichette : Etichette[0...N]) : Attività[0...N]	<ul style="list-style-type: none"> nella barra di ricerca del titolo non si possono inserire più di 20 caratteri i due valori della durata media sono compresi tra 0 e 999, sono interi e il primo è minore del secondo il numero di partecipanti non può superare 99 	il catalogo viene filtrato secondo le etichette previste
creaAttività(attività : Attività)	<ul style="list-style-type: none"> la descrizione non può superare i 2000 caratteri i due valori della durata media sono compresi tra 0 e 999, sono interi e il primo è minore del secondo il numero di partecipanti non può superare 99 il titolo non può superare i 20 caratteri di lunghezza 	viene aggiunta una nuova attività al catalogo

```
context Catalogo::aggiornaCatalogo()
post: self.ultimoAggiornamento = Data.now()
```

```
// TODO
```

ListaAttività

Metodo	Precondizioni	Postcondizioni
creaLista() : listaAttività	<ul style="list-style-type: none"> il nome della lista di attività non può superare i 20 caratteri di lunghezza un utente non può creare più di 99 liste di attività il nome della lista di attività non può essere uguale al nome di un'altra lista già presente il nome della lista di attività non può essere nessuno 	viene creata una nuova lista di attività
aggiungiAttività(attività : Attività)	il numero di attività in una lista non può superare 9999	l'attività scelta viene aggiunta alla lista

esporta(formato : String) : File		la lista viene esportata in formato pdf o json
eliminaAttività(indice : int)		l'attività con l'indice scelto viene rimossa dalla lista

// TODO

Attività

// TODO: Invarianti // titolo non vuoto e univoco

Metodo	Precondizioni	Postcondizioni
modifica(attivitàModificata : Attività)	<ul style="list-style-type: none"> la descrizione non può superare i 2000 caratteri i due valori della durata media sono compresi tra 0 e 999, sono interi e il primo è minore del secondo il numero di partecipanti non può superare 99 il titolo non può superare i 20 caratteri di lunghezza 	<ul style="list-style-type: none"> tutti gli attributi assumono il valore dell'attività modificata l'attributo ultimaModifica assume il valore della data corrente

Segnalazione

// TODO: invarianti

Metodo	Precondizioni	Postcondizioni
inviaSegnala(autore : Utente, attività : Attività, voto : String)	l'attributo messaggio non può superare i 500 caratteri di lunghezza	viene aggiunta una segnalazione per l'attività scelta

Valutazione

// TODO: invarianti

Metodo	Precondizioni	Postcondizioni
inviaValutazione(autore : Utente, attività : Attività, voto : String)	il voto inserito deve essere un numero decimale compreso tra 0 e 5, con scarto di 0.5	<ul style="list-style-type: none"> viene aggiunta la valutazione all'attività scelta cambia la media di voti dell'attività scelta

GestoreDatiOffline

Metodo	Precondizioni	Postcondizioni
--------	---------------	----------------

MongoDB

Metodo	Precondizioni	Postcondizioni
--------	---------------	----------------

Info

Invarianti:

titolo : String

- il titolo deve avere al massimo 20 caratteri

età : Tuple{da : int, a : int}

- il range di età deve avere estremi compresi tra 0 e 100 esclusi

durata : Tuple{da : int, a : int}

- il range di durata deve avere estremi compresi tra 0 e 1.00 esclusi

giocatori : Tuple{da : int, a : int}

- il range di giocatori deve avere estremi compresi tra 0 e 100 esclusi

giocatoriPerSquadra : int

- il numero di giocatori per squadra dev'essere compreso tra 0 e 100 esclusi

numeroSquadre : int

- il numero di squadre dev'essere compreso tra 0 e 100 esclusi

```
context Info inv:
self.titolo.size() <= 20
```

```
context Info inv:
0 < self.età.da AND self.età.da <= self.età.a AND self.età.a < 100
```

```
context Info inv:
0 < self.durata.da AND self.durata.da <= self.durata.a AND self.durata.a < 1.000
```

```
context Info inv:
0 < self.giocatori.da AND self.giocatori.da <= self.giocatori.a AND self.giocatori.a < 100
```

```
context Info inv:
0 < self.giocatoriPerSquadra AND self.giocatoriPerSquadra < 100
```

```
context Info inv:
0 < self.numeroSquadre AND self.numeroSquadre < 100
```

Filtro

Invarianti:

giocatori : Tuple{da : int, a : int}

- il range del numero di giocatori deve essere costituito da un solo valore (l'attuale numero di giocatori presenti)

mediaMinima : int

- la media minima dev'essere compresa tra 0 e 10 inclusi

numeroSegnalazioniMinimo : int

- il numero minimo di segnalazioni dev'essere compreso tra 0 e 9.999 inclusi

```
context Filtro inv:
self.giocatori.da = self.giocatori.a
```

```
context Filtro inv:
0 <= self.mediaMinima AND self.mediaMinima <= 10
```

```
context Filtro inv:
0 <= self.numeroSegnalazioniMinimo AND self.numeroSegnalazioniMinimo < 10.000
```

Faccia

Invarianti:

parola : String

- la parola può avere al massimo 20 caratteri e, se il tipo è parole, non deve essere vuota

```
context Faccia inv:
self.parola.size() < 20 AND (self.tipo = "parole" implies self.parola <> "")
```

Colore

Invarianti:

R : int

- il valore del colore rosso dev'essere compreso fra 1 e 255 inclusi

G : int

- il valore del colore verde dev'essere compreso fra 1 e 255 inclusi

B : int

- il valore del colore blu dev'essere compreso fra 1 e 255 inclusi

```
context Colore inv:
(0 <= R AND R < 256) AND (0 <= G AND G < 256) AND (0 <= B AND B < 256)
```

URL

Invarianti:**protocollo : String**

- il protocollo dev'essere http o https o file

```
context URL inv:
self.protocollo = "http" OR self.protocollo = "https" OR self.protocollo = "file"
```

Data

Invarianti:**giorno : int**

- il giorno dev'essere compreso tra 1 e 31 (inclusi), escluso 31 se il mese è aprile, giugno, settembre o novembre, esclusi 30 e 31 se il mese è febbraio, escluso anche 29 se il mese è febbraio e l'anno non è bisestile

mese : int

- il mese dev'essere compreso tra 1 e 12 (inclusi)

anno : int

- l'anno dev'essere compreso tra 1 e 9.999 (inclusi)

orario : Time

- le ore devono essere minori di 24

```
context Data inv:
0 < self.giorno AND self.giorno <= 31 AND (Set{2, 4, 6, 9, 11}->count(self.mese) > 0 implies
self.giorno <> 31) AND (self.mese = 2 implies (self.giorno <> 30 AND if ((self.anno % 4 <> 0) OR
(self.anno % 100 = 0 AND self.anno % 1000 <> 0)) then self.giorno <> 29 endif))
```

```
context Data inv:
0 < self.mese AND self.anno <= 12
```

```
context Data inv:
0 < self.anno AND self.anno < 10.000
```

```
context Data inv:
self.orario.ore < 24
```

Time

Invarianti:

ore : int

- le ore devono essere un intero non negativo

minuti : int

- i minuti devono essere compresi fra 0 (incluso) e 60 (escluso)

secondi : int

- i secondi devono essere compresi fra 0 (incluso) e 60 (escluso)

centesimi : int

- i centesimi devono essere compresi fra 0 (incluso) e 100 (escluso)

```
context Time inv:
0 <= self.ore
```

```
context Time inv:
0 <= self.minuti AND self.minuti < 60
```

```
context Time inv:
0 <= self.secondi AND self.secondi < 60
```

```
context Time inv:
0 <= self.centesimi AND self.centesimi < 100
```

Etichetta

Invarianti:

nome : String

- i nomi delle etichette devono essere univoci, non vuoti e avere massimo 20 caratteri

descrizione : String

- le descrizioni delle etichette devono avere al massimo 2000 caratteri

categoria : String

- i nomi delle categorie devono essere non vuoti e avere massimo 20 caratteri

```
context Etichetta inv:
Etichetta.allInstances()->forAll(e1, e2 |
```

```
e1 <> e2 implies e1.nome.toLowerCase() <> e2.nome.toLowerCase() AND self.nome <> "" AND  
self.nome.size() <= 20
```

```
context Etichetta inv:  
self.descrizione.size() <= 2000
```

```
context Etichetta inv:  
self.categoria <> "" AND self.categoria.size() <= 20
```

Metodo	Precondizioni	Postcondizioni
creaEtichetta(richiedente : Utente, nome : String, descrizione : String, categoria : String)	il richiedente dev'essere un amministratore	

```
context Etichetta::creaEtichetta(richiedente : Utente, nome : String, descrizione : String,  
categoria : String)  
pre: richiedente.ruolo = "admin"  
post:
```