Jolf
Relazione per Programmazione ad Oggetti

Lorenzo Zanetti
Fabio Pedrini
Alessandro Zanzi

25 ottobre 2021

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il progetto si pone come obiettivo quello di realizzare un videogioco di minigolf con grafica 2D, visuale dall'alto e controlli di tipo click & drag. All'interno del gioco sarà possibile selezionare diverse mappe, in ciascuna di queste mappe saranno presenti 3 stelle (oltre ad altri ostacoli, come muri o sabbia) e una volta che la pallina attraversa 2 di queste stelle la terza diventa la buca. L'obiettivo resta quello di utilizzare meno tiri possibili, motivo per cui verrà salvata una classifica con i migliori punteggi. In seguito "livello" e "mappa" verranno utilizzati come sinonimi poichè concetti molto simili, infatti nel nostro caso superare un livello significa "risolvere" una mappa, cioè riuscire a fare arrivare la pallina in buca (non è però garantito che nell'implementazione i due concetti non vengano separati).

Requisiti funzionali

- All'apertura dell'applicazione dovrà presentarsi un menù dal quale poter iniziare una partita, selezionare un livello, vedere la classifica o uscire (all'inizio di una partita verrà chiesto un nome con il quale essere registrati nella classifica).
- Iniziando una partita dovranno presentarsi in successione tutte le mappe del gioco (o in alternativa una collezione di esse, nel caso ne siano state implementate molte), il giocatore dovrà superarle una per una e alla fine il suo punteggio (i tiri totali effettuati) verrà salvato nella classifica.

- Premendo il tasto "livelli" invece si ha la possibilità di giocare un solo livello a scelta, con la finalità di esercitarsi su una mappa in particolare, e quindi senza salvare il punteggio.

- Durante lo svolgimento di una partita dovrà essere possibile "colpire" la pallina solo quando questa è ferma; essa, una volta "colpita", dovrà quindi iniziare a muoversi ed interagire in modo coerente con ogni oggetto sul suo percorso (es. rimbalzare contro un muro) fino a quando si fermerà e sarà necessario un nuovo colpo o fino a quando non entrerà in buca.

- Come descritto nell'introduzione la buca non dovrà essere subito presente nella mappa, ma dovranno esserci sempre 3 stelle. Una volta che la pallina ne attraversa 2 (le stelle spariscono quando attraversate) apparirà una buca in corrispondenza della terza, facendola sparire.

- La pallina potrà essere tirata tenendo premuto con il mouse su un punto qualunque della mappa, trascinandolo e lasciando. Durante questo processo apparirà sulla pallina una freccia che indicherà direzione e intensità del tiro (in verso opposto rispetto a dove si trascina il mouse) e potrà essere modificata fino al rilascio.

Requisiti non funzionali

- Il movimento della pallina e le interazioni con gli oggetti della mappa vanno gestititi efficientemente in modo che la resa video sia fluida.

- Il sistema di lancio della pallina deve risultare responsivo

1.2 Analisi e modello del dominio

Il giocatore dovrà essere in grado, attraverso un menù, di poter giocare un determinato percorso scelto da una lista di percorsi predefiniti. Ogni singolo percorso è composto da una serie ordinata di mappe, che verranno svolte in successione. Ciascuna mappa ha dimensioni proprie (larghezza e altezza, ma sempre rettangolare) ed è composta da una serie di elementi. Tra questi elementi i principali saranno una pallina e tre "stelle", mentre saranno opzionali e caratteristici di ogni mappa la prensenza di ostacoli fisici, superfici con diverso attrito ed eventualmente ostacoli in movimento. Per completare la mappa il giocatore dovrà colpire con la pallina due delle tre stelle e portare la pallina all'interno della buca, che apparirà in corrispondenza della terza stella non ancora colpita. Lo svolgimento del percorso si conclude quando

l'utente completa tutte le mappe di quel percorso. In alternativa deve essere possibile (sempre tramite il menù) giocare una sola mappa scegliendola sempre da quelle preimpostate, che in generale coincideranno con quelle presenti nei percorsi. Il diagramma UML che descrive questi elementi è mostrato in Figura 1.1.

La difficoltà primaria sarà quella di riuscire a ottimizzare le interazioni tra gli oggetti della mappa mantenendo dei comportamenti corretti ed accurati, che richiederebbe diversi test e implementazione di algoritmi efficienti. Per cui si cercherà di raggiungere il risultato migliore all'interno del monte ore previsto senza dover trascurare altri aspetti del progetto. Sempre per questo motivo la realizzazione di ostacoli in movimento sarà fatta solo successivamente ad una versione funzionante dell'applicativo privo di essi, se saranno rimaste ore sufficienti. I requisiti che dovranno, invece, essere affrontati obbligatoriamente sono: un modo per permettere all'utente (e quindi all'interfaccia) di poter accedere alle mappe singolarmente e alle mappe che compongono un determinato percorso e il regolare svolgimento di una mappa "semplice", che contiene quindi elementi essenziali (assenza di ostacoli complessi, per esempio in movimento).

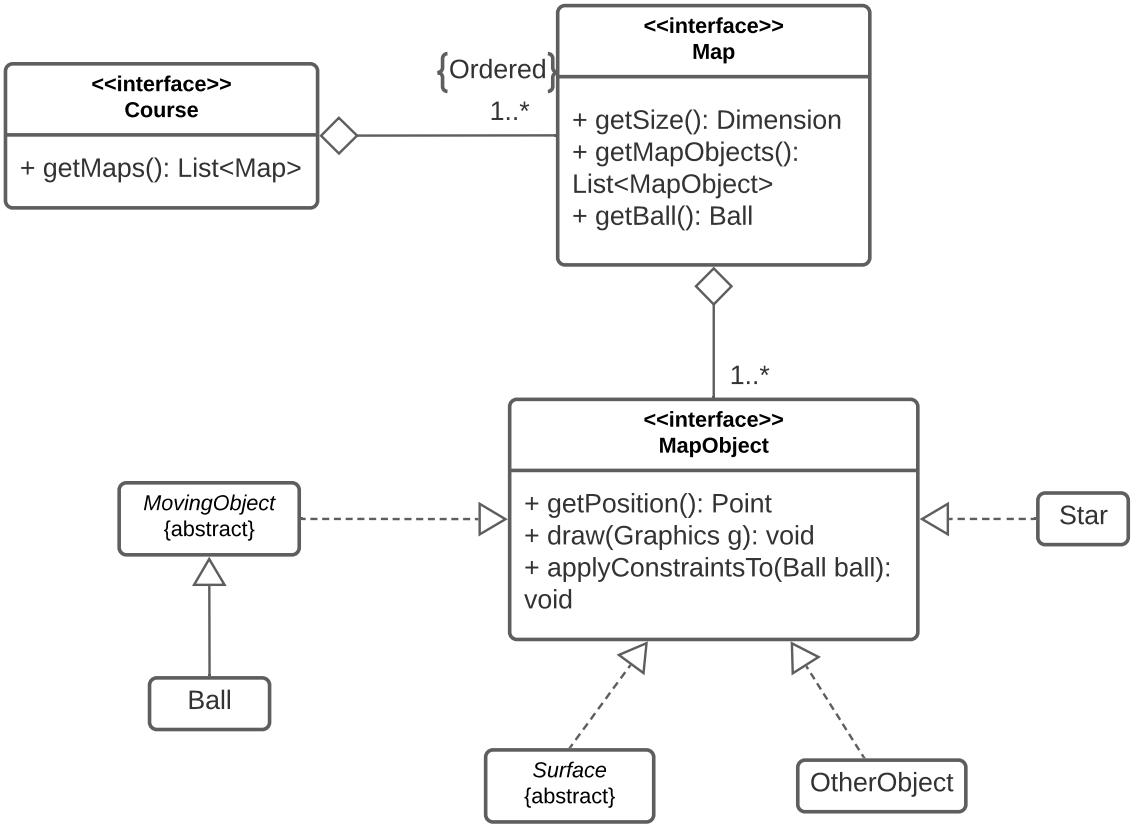

Figura 1.1: Schema UML sul modello del problema

Capitolo 2

Design

2.1 Architettura

L'architettura del progetto segue per la maggior parte il pattern architetturale MVC, con alcune piccole differenze. Più nello specifico, a livello architetturale, si è deciso di suddividere l'intefaccia View in due diverse interfacce, una per l'input e una per l'output, per differenziare i due compiti che normalmente verrebbero svolti entrambi dall'interfaccia View. La differenza che comporta questo approccio da MVC vero e proprio è che il controller dovrà collegarsi con due diverse classi, una che implementa le operazioni di input e una che implementa quelle di output, o come nel nostro caso con una sola classe che le implementa entrambe. Qual è quindi il vantaggio di questa scelta? In un futuro si potrà per esempio cambiare la classe che implementa l'input dell'applicazione lasciando l'output inalterato o viceversa. In particolare il controller ha il compito di gestire lo svolgimento della "partita" che può essere un percorso o una sola mappa, durante tutta la durata della partita aggiornerà il suo output, comunicandogli gli oggetti che si trovano sulla mappa, i tiri effettuati, la dimensione della mappa e l'eventuale conclusione della partita. Allo stesso tempo dovrà comunicare all'input quando è possibile effettuare un nuovo tiro, cioè quando la pallina si ferma. L'interfaccia di input sarà quindi in grado di essere utilizzata dall'utente solo una volta che è abilitata dal controller e, una volta che l'utente effettua un tiro, dovrà comunicarlo al controller e tornare in stand-by, in attesa che venga attivata dal controller nuovamente. Come è facile capire l'interfaccia di output verrà "utilizzata" dal controller per comunicare al giocare come si sta svolgendo il gioco e non avrà altre funzionalità. Infine gli oggetti della mappa, quindi il "model", interagiranno tra di loro e basta, mentre verranno "osservati e controllati" dal controller appunto. Tutti questi elementi dell'applicazione

verranno gestiti in primo luogo da una GUI iniziale che guiderà l'utente nella scelta delle mappe, dei percorsi, nella visualizzazione della classifica e si occuperà di "dare il via" al gioco vero e proprio.

In Figura 2.1 è esemplificato il diagramma UML architetturale.

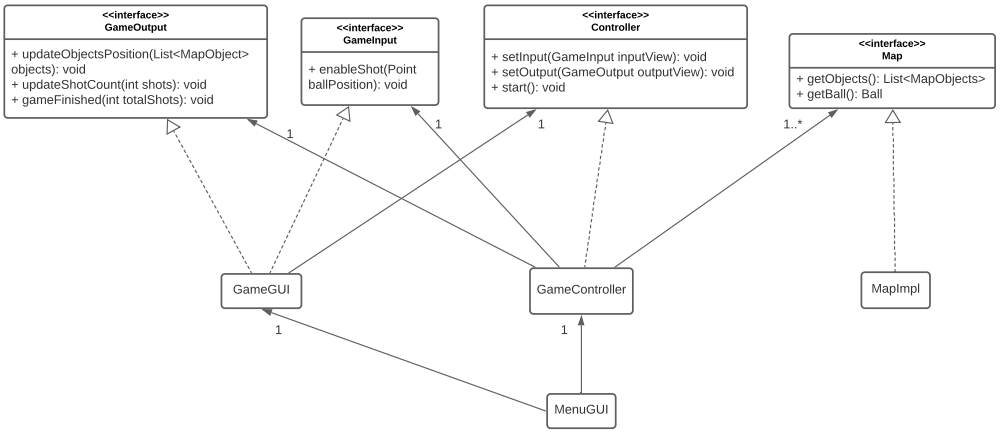

Figura 2.1: Schema UML architetturale di Jolf. La classe MenuGUI è la nostra applicazione iniziale, GameOutput e GameInput sono le rispettive due parti della View, GameController il controller e Map e una parte del nostro Model di cui si serve GameController. In fase di progettazione saranno aggiunte più funzioni e più collegamenti, per esempio tra controller e model, ma preferiamo mantenere lo schema più riassuntivo e leggibile

2.2 Design dettagliato

2.2.1 Sezione di progetto di Zanetti Lorenzo

Navigazione del Menù

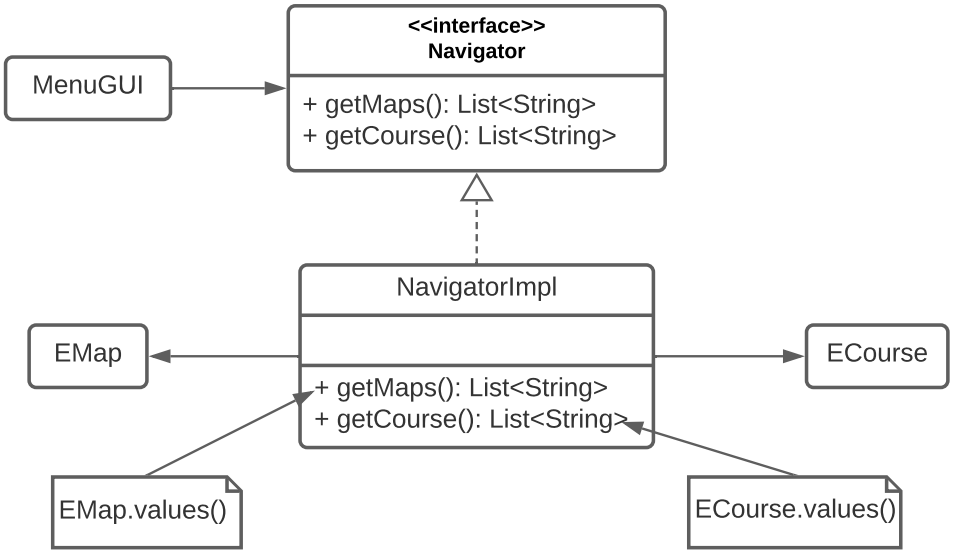

Figura 2.2: Rappresentazione UML del pattern Adapter usato per adattare mappe e percorsi all'uso della GUI del menù

Problema La classe che sviluppa il menù iniziale dell'applicazione deve gestire la visualizzazione degli elenchi delle mappe e dei percorsi, ma vogliamo evitare che questa debba conoscere con precisione l'implementazione del modello per consentire riusabilità e rispettare il pattern architetturale MVC.

Soluzione Per consentire la massima riusabilità utilizzeremo il pattern adapter, infatti la classe MenuGUI mantiene al suo interno un rifermimento ad un oggetto che implementa l'interfaccia Navigator, a cui può "chiedere" delle liste di stringhe che contengono i nomi delle mappe e quelli dei percorsi. La classe NavigatorImpl che implementa questa interfaccia ha quindi il compito di adattare gli oggetti del modello (cioè EMap e ECourse) in modo che siano di facile comprensione alla GUI.

Traduzione Nel nostro caso il Client è la MenuGUI, il target delle sue chiamate è Navigator, la sua realizzazione NavigatorImpl è l'adapter e i due diversi adaptee sono EMap e ECourse.

Problemi simili Questo pattern risolve un problema comune in maniera molto semplice, per cui all'interno del codice verrà utilizzato altre volte in contesti diversi. Per esempio, per quanto riguarda la mia parte viene usato spesso nelle GUI con delle innerclass che estendono Listener di java.awt all'interno di MenuGUI o di InputPanel. Nello schema UML di Figura 2.2 è stato rappresentato solo il "caso Navigator" perchè rappresentabile in modo più chiaro ed esemplificativo non essendo una innerclass.

Oggetti della mappa in movimento

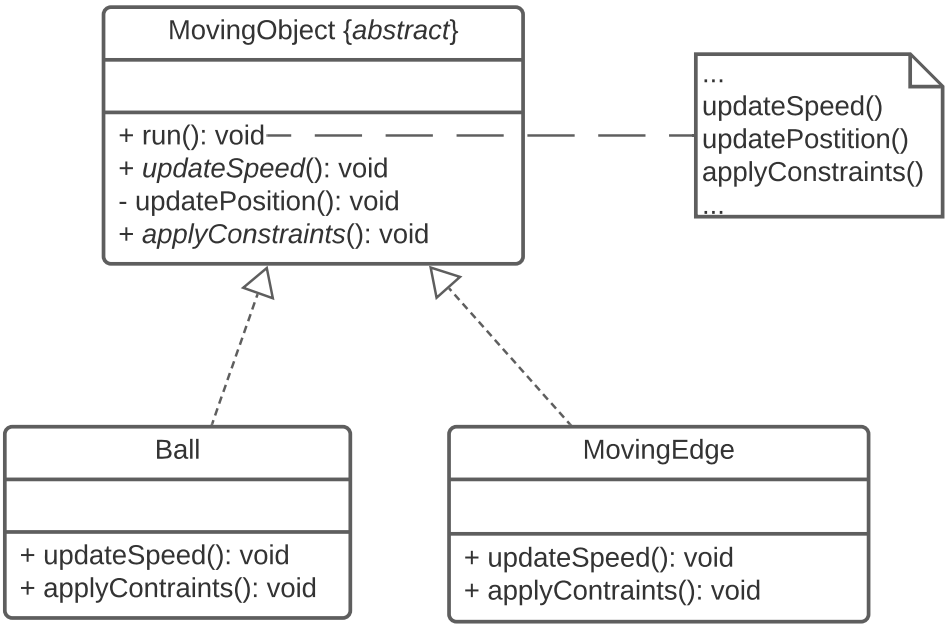

Figura 2.3: Rappresentazione UML del pattern template method usato per definire l'algoritmo di movimento degli oggetti in maniera generale, per poi specializzarlo nelle sottoclassi

Problema Nell'applicazione saranno presenti ora o in futuro altri oggetti diversi dalla pallina che si muovono nella mappa, per questo motivo dobbiamo

cercare di implementare il codice in maniera che sia riutilizzabile. Una cosa importante da notare è che non si muoveranno nello stesso modo, infatti: la pallina dovrà muoversi rimbalzando contro oggetti fisici e rallentando fino a fermarsi, mentre un eventuale muro che fa da ostacolo dovrà per esempio andare su e giù da un estremo all'altro della mappa con una velocità costante.

Soluzione Per fare ciò utilizziamo il pattern comportamentale template method (mostrato in Figura 2.3) che permette di definire una classe astratta con un algoritmo scheletro, generico, che gestisca il movimento di un oggetto qualsiasi. Nel fare ciò definiamo una parte di algoritmo comune a qualunque oggetto in movimento, per esempio il calcolo della posizione al passare di un tempo t con una velocità v. Un'altra parte dell'algoritmo è invece costituita da dei metodi astratti e non definiti, che nel nostro caso gestiscono la variazione della velocità al passare del tempo o il comportamento dato dalle collisioni. Questi due metodi vengono poi definiti nelle due sottoclassi che descrivono il comportamento della pallina e di un muro in movimento.

Traduzione Nel nostro caso il template method è run(), che gestisce l'update della posizione dell'oggetto e al suo interno ha le due operazioni primitive updateSpeed() e applyConstraints(). Di conseguenza la classe astratta è MovingObject e le classi concrete sono Ball e MovingEdge.

Note su questo problema Il problema inizialmente non riguardava la mia parte di progetto ma quella di Pedrini (movimento pallina), ma per il fatto che una delle mie parti (collisioni pallina) era particolarmente collegata abbiamo svolto la parte di design e sviluppo di queste classi in collaborazione.

Controller per giocare un percorso o una mappa

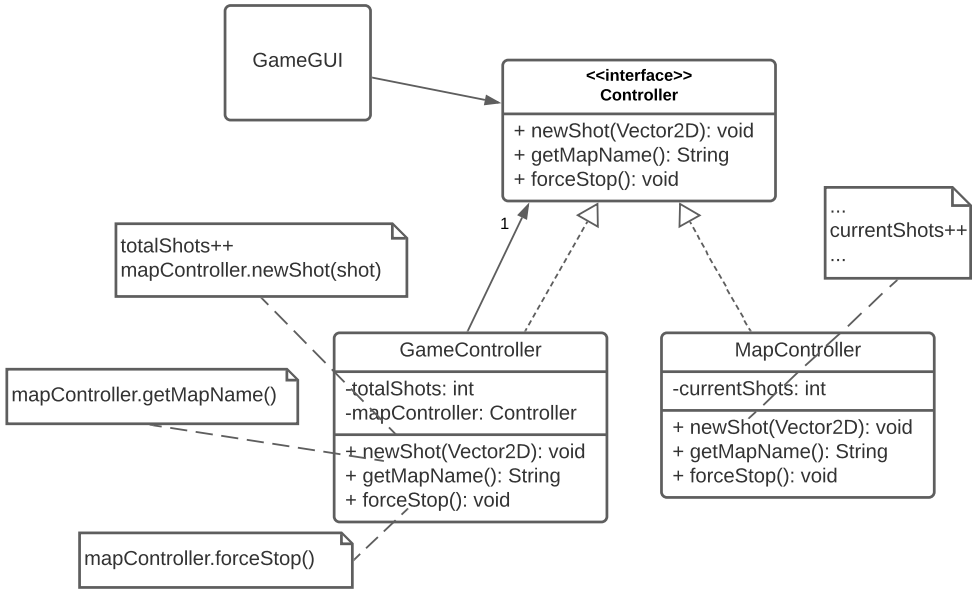

Figura 2.4: Rappresentazione UML del pattern decorator usato per definire due diversi tipi di Controller in cui il GameController utilizza il MapController per svolgere i compiti "base" e lo arricchisce con nuove funzionalità

Problema Nell'applcazione vogliamo gestire due tipi differenti, ma simili, di "partita" che verranno visualizzate dalla stessa GUI (GameGUI). "Giocando" un percorso, si giocano più mappe, si inserisce un nome con il quale finire in classifica e ovviamente si conteggiano i tiri complessivi tra le diverse mappe, oppure, si può giocare una sola mappa e in questo caso le funzionalità precedentemente citate vengono a meno. N.B. In questo schema per semplicità sono state messe come esempio solo tre metodi, ma il controller ne ha di più e ciascuna viene comunque delegata da GameController a MapController e/o arricchita.

Soluzione Per fare ciò in modo efficiente, senza dover ripetere codice inutilmente, utilizziamo il pattern Decorator (mostrato in Figura 2.4) che ci permette di definire una classe MapController, che implementa l'interfaccia Controller, che gestisce lo svolgimento di una mappa, sarà in grado di comunicare gli elementi presenti nella mappa alla GUI, di contare i tiri effettuati

in questa mappa e di controllarne il normale svolgimento. Fatto ciò creiamo un'altra classe che implementa questa interfaccia (GameController) che al suo interno ha un riferimento ad un MapController per gestire lo svolgimento di un percorso, il nostro decorator (sempre GameController) scorrerà le mappe una per una delegando il loro svolgimento al MapController e occupandosi solo di arricchire il suo comportamento, salvando un nome giocatore, contando i tiri totali ecc...

Traduzione Nel nostro caso il Client è GameGUI, il Component è Controller che viene esteso dai component concreti MapController e GameController, quest'ultimo è il Decorator.

Applicazione dei vincoli alla pallina

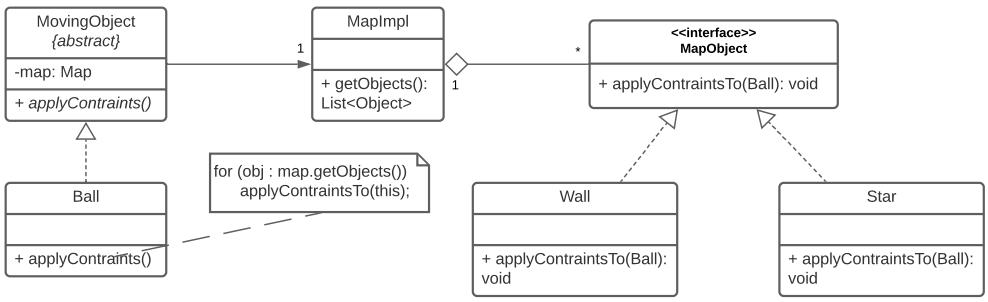

Figura 2.5: Rappresentazione del pattern Bridge utilizzato per permettere alla pallina di delegare il compito di controllare le interazioni con gli oggetti della mappa agli oggetti stessi

Problema Nello svolgimento di una mappa l'elemento più importante di cui vogliamo tenere traccia è la pallina. Questo elemento è in costante movimento e ogni manciata di millisecondi deve quindi controllare se sta toccando un muro e quindi rimbalzare, se sta attraversando una zona con diverso attrito ecc. Come possiamo realizzare il codice in modo da non rendere la classe che gestisce la pallina un lunghissimo elenco di metodi, caratteristici di ogni altro oggetto della mappa e senza doverla modifica all'aggiunta di una nuova classe che implementa MapObject?

Soluzione Per fare ciò abbiamo deciso di utilizzare (in modo leggermente diverso) il pattern Bridge. In particolare, la classe Ball (che implementa MovingObject) mantiene un rifermento alla mappa a cui appartiene e grazie

ad esso può reperire ogni oggetto presente sulla mappa e delegare ad essi il compito di controllare se ci sono collisioni, intersezioni, se la pallina si trova all'interno di zone particolari della mappa ecc. In questo modo la pallina non dovrà interessarsi di quali siano le specifiche interazioni o dell'aggiunta di nuove classi con cui dovrà interagire, ma sarà compito dei singoli oggetti.

Traduzione Nel nostro caso Ball è la RefinedAbstraction dell'Abstraction MovingObject, MapObject è l'Implementor e le sue implementazioni (come Wall o Star) sono i ConcreteImplementor. Nel pattern Bridge non sarebbe quindi presente la classe MapImpl, ma l'unica differenza è che MovingObject avrebbe direttamente un riferimento ad una lista di MapObject, per cui MapImpl non è altro che uno step aggiuntivo al "ponte" che collega Abstraction ad Implementor.

2.2.2 Sezione di progetto di Pedrini Fabio

Creazione degli ostacoli

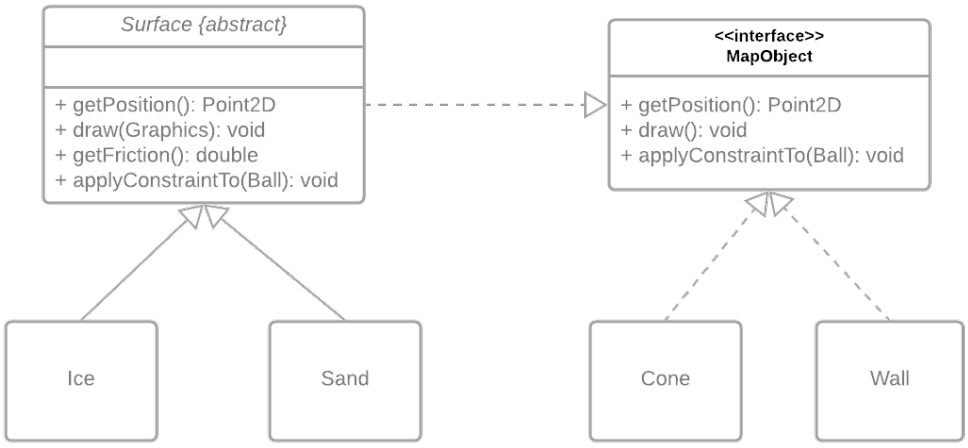

Figura 2.6: Rappresentazione UML della creazione di ostacoli statici.

Problema Gli ostacoli da inserire nelle mappe sono di varie tipologie (ad esempio: oggetti contro cui si deve rimbalzare, oggetti in movimento, oggetti "attraversabili" con diverso attrito...) e si vuole definire uno scheletro o un comportamento comune fra di essi.

Problemi simili Il problema è molto simile a quello trattato da Zanetti nella gestione degli oggetti della mappa in movimento; in questa sezione vengono trattati gli oggetti statici.

Soluzione Per rappresentare un oggetto della mappa in generale è stata creata l'interfaccia MapObject la quale viene implementata dalla classe astratta Surface utilizzando il pattern Template Method. All'interno della classe Surface il template method è il metodo applyConstraintTo() che calcola l'attrito da assegnare alla pallina in base alla superficie in cui si trova. Nel nostro caso specifico il metodo getFriction() (da cui viene ricavato l'attrito) risulta essere banale in quanto ritorna una costante che varia in base alla superficie, ma in futuro potrebbero essere aggiunte nuove superfici in cui l'attrito viene calcolato con algoritmi complessi e in questo caso il pattern Template Method risulta particolarmente efficace.

Lettura e scrittura della classifica

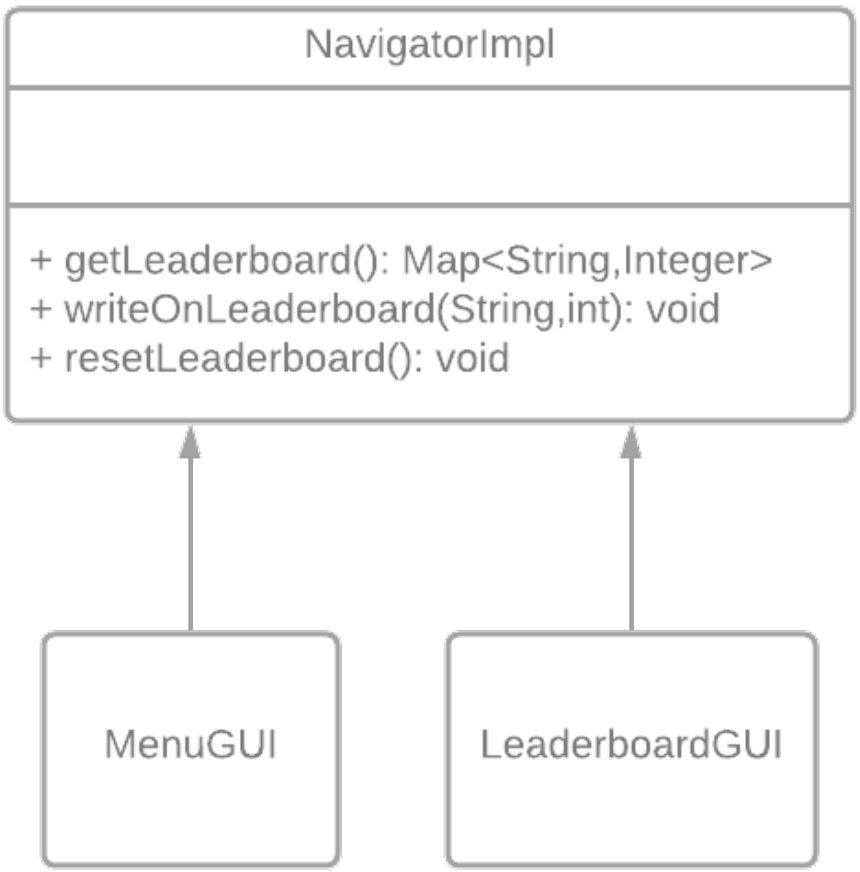

Figura 2.7: Rappresentazione con diagramma UML delle classi coinvolte nella lettura e scrittura nella classifica.

Problema Alla fine di ogni percorso è necessario salvare su file il nome e il punteggio che il giocatore ha appena ottenuto. Si vuole impedire agli elementi della view di scrivere direttamente su file, in quanto non è un compito destinato ad essi.

Soluzione La lettura del file contenente i dati della classifica viene richiesta esplicitamente dalla classe LeaderboardGUI che dovrà far visualizzare i dati all'utente. La richiesta viene effettuata grazie ad un campo di tipo Navigator presente nella GUI: è infatti questa classe del controller che si occupa della lettura (e anche della scrittura) della classifica. Per leggere il file viene

usato un BufferedReader che legge le linee del file, il navigator colleziona i dati in una mappa (java.util.Map) e li ritorna alla GUI. La classe NavigatorImpl utilizza un BufferedWriter per scrivere su file che aggiunge una riga senza sovrascrivere il contenuto del file. Al contrario, quando la classifica va resettata, il BufferedWriter sovrascrive tutto quello che era presente sul file leaderboard.txt.

Creazione delle mappe

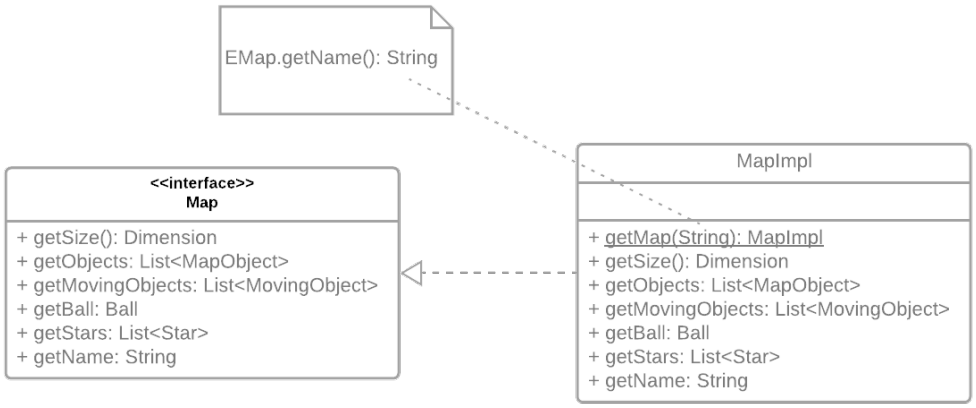

Figura 2.8: Rappresentazione UML della Static Factory utilizzata per creare le mappe.

Problema All'interno dell'applicazione devono essere create diverse mappe (o buche) che non hanno sempre lo stesso numero di elementi e si vuole evitare che la creazione sia divisa in più classi, una per ciascuna mappa.

Soluzione Per creare le mappe del gioco è stata sviluppata la classe MapImpl al cui interno vengono istanziati tutti gli oggetti presenti nell'enumerazione EMap. La classe MapImpl è di fatto una Static Factory: il costruttore è statico e al suo interno vengono inseriti tutti i dati utili per la definizione di una mappa (quali ostacoli sono presenti e in quale posizione si trovano).

Note sul problema Era stata presa in considerazione l'idea di utilizzare un pattern Builder per rendere più leggibile il processo di creazione ma si è optato per implementare una Static Factory dato che entrambi i pattern sono ugualmente efficaci.

2.2.3 Sezione di progetto di Zanzi Alessandro

Funzionalità del menù principale

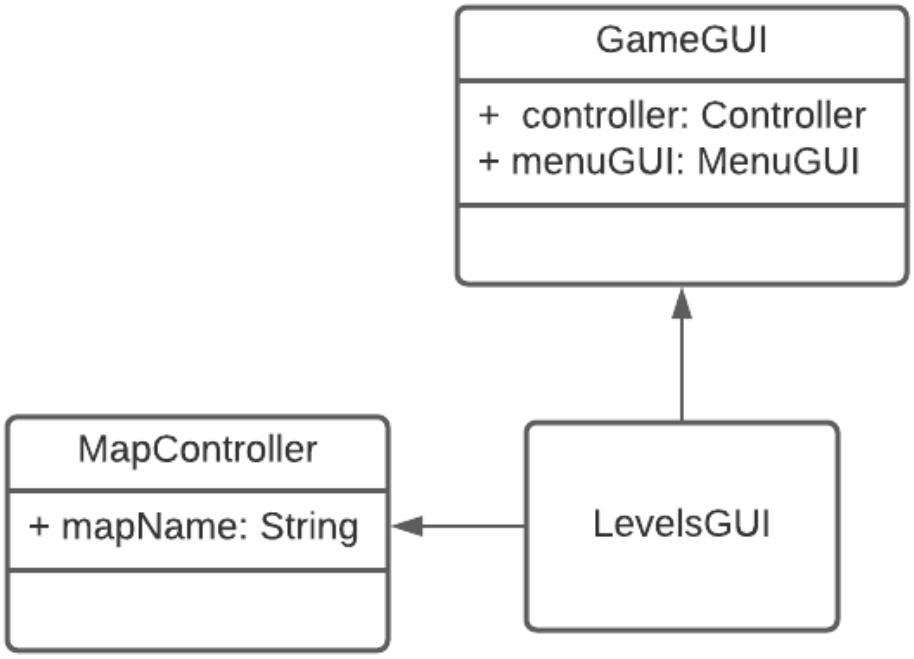

Figura 2.9: Rappresentazione con diagramma UML delle classi coinvolte nella gestione dei vari livelli.

Problema Nel menù principale sono implementati vari bottoni usati dall'utente per scegliere quali azioni svolgere. Si vuole dare funzionalità specifiche a ciasun bottone. Prendendo per esempio in considerazione il bottone LEVELS, si vuole realizzare un bottone per ciascuna mappa, senza crearne uno nuovo ogni volta, in modo che si usino tutte le mappe create, e nel caso se ne aggiungano di nuove sia facile inserirle.

Soluzione Per risolvere il problema ho sfruttato un foreach che crea un nuovo bottone per ogni mappa presente crea un nuovo JButton al quale affianca un Listener, il quale, una volta premuto il bottone, fa partire una nuova GameGUI e un MapController a cui è passata la mappa corrente.

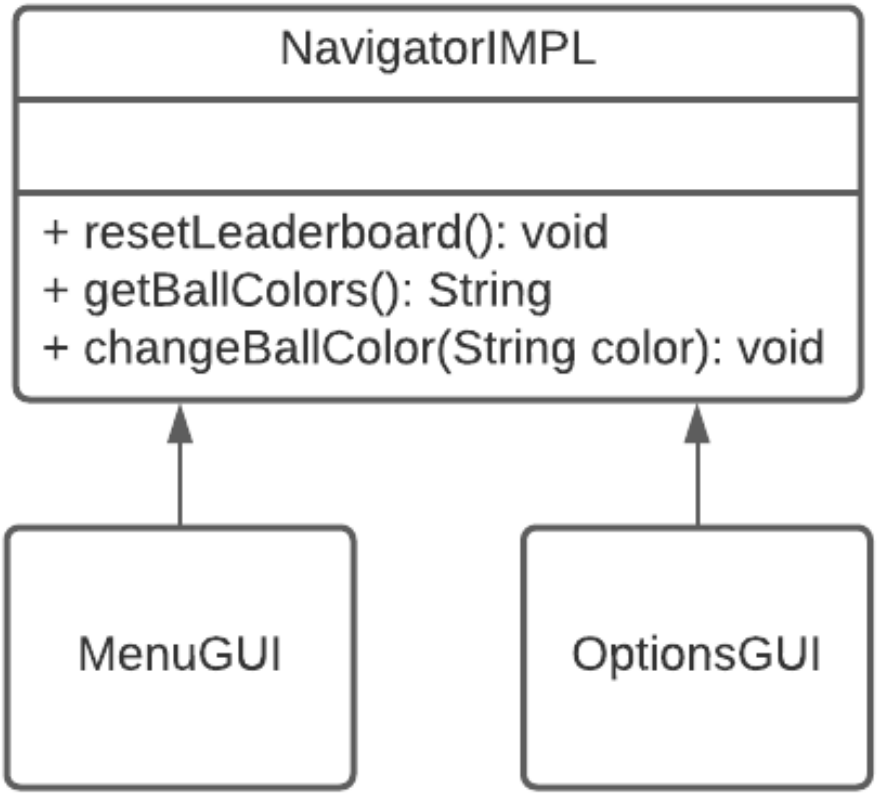

Figura 2.10: Rappresentazione con diagramma UML delle classi coinvolte nella gestione delle varie opzioni.

Problema Per il bottone OPTIONS si vuole dare la possibilità di scegliere il colore della pallina e di azzerare la classifica.

Soluzione Per quanto riguarda il colore della pallina, si è optato per l'utilizzo di una select su cui è posto un actionListener che una volta scelto il colore lo passa al metodo changeBallColor di NavigatorImpl e cambia il colore. Anche per il reset della leaderboard si sfrutta un metodo di NavigatorImpl, in particolare resetLeaderboard.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il testing automatizzato è stato realizzato con l'utilizzo di JUnit 5 per il controllo automatico del funzionamento di alcune classi specifiche come Angle, Vector2D e Point2D. Non è stato invece utilizzato questo approccio per classi come Ball o MovingEdge che basano il loro funzionamento su Thread e modificano la loro posizione ogni istante, basandosi su velocità, accelerazione e altri fattori, per cui sarebbe stato molto complesso verificare il comportamento di queste classi con test automatici e in ogni caso non era nostro obiettivo raggiungere una precisione millimetrica negli spostamenti degli oggetti su schermo bensì avere un risultato "realistico". Queste classi sono state oltretutto realizzate successivamente alle GUI che li visualizzano, per cui è stato molto più semplice e veloce testarli sul campo e visivamente con test manuali. Anche per quanto riguarda le GUI non abbiamo ritenuto proficuo realizzarne dei test automatizzati. Per esempio la GUI "MenuGUI" ha come comportamento quello di chiudersi (o meglio diventare invisibile) quando viene cliccato uno qualunque dei suoi tasti ed aprire la GUI successiva, comportamento motlo difficile da testare con JUnit e molto più semplice da testare visivamente. Sono comunque stati fatti test manuali, più approssimativi ma veloci, sulla corretta visualizzazione degli oggetti o sull'utilizzo dell'input (esempio nel main commentato della classe "TestClickAndDrag").

3.2 Metodologia di lavoro

Come da raccomandazioni un membro del gruppo si è preso l'incarico di creare il repository GitHub con una struttura iniziale. Sono stati inizialmente discussi in gruppo gli aspetti architetturali e del modello dell'applicazione,

sono stati realizzati gli schemi UML e ci si è suddivisi il lavoro come pianificato originariamente sul forum. Ogni membro ha collaborato allo sviluppo facendo pull dal repository remoto e lavorando alle proprie parti in locale. Ogni membro, quando l'ha ritenuto necessario, ha condiviso il proprio codice facendo push e nella maggior parte dei casi non è stato necessario compiere merge manuali perchè si lavorava su parti diverse di codice. Quando è stato necessario collaborare a classi che intersecavano l'ambito di sviluppo di 2 o più membri si sono svolte parti di codice in collaborazione, da un solo terminale o si sono risolti eventuali conflitti merge manualmente, in accordo con gli altri membri.

Lorenzo Zanetti

- Collaborazione allo sviluppo di tutte le classi del package "controller"
- Collaborazione allo sviluppo di tutte le classi del package "util"
- Collaborazione allo sviluppo di tutte le classi del package "tests"
- Collaborazione allo sviluppo di tutte le classi (esclusa "Cone") del package "model"
- Sviluppo delle classi e interfacce "AppMain","MenuGUI","ShotInput", "ShotListener","ShotVisualizer", "InputPanel" del package "view"
- Collaborazione allo sviluppo delle classi e interfacce "GameOutput", "GameInput", "GameGUI", "StartingGUI" del package "view"

Fabio Pedrini

- Collaborazione allo sviluppo di tutte le classi del package "model"
- Collaborazione allo sviluppo di parte delle classi del package "view"
- Collaborazione allo sviluppo dell'interfaccia "Navigator" e della classe "NavigatorImpl" del package "controller"
- Collaborazione allo sviluppo delle classi "MyOptionPane" e "MyTitle" del package "util"

Alessandro Zanzi

- Collaborazione allo sviluppo di tutte le classi del package "view"
- Collaborazione allo sviluppo di parte delle classi del package "model"
- Sviluppo delle classi e interfacce "LevelsGUI", "MenuGUI", "OptionGUI", "LeaderboardsGUI", "StartingGUI" del package "view"
- Collaborazione allo sviluppo delle classi e interfacce "Map", "MapImpl", "Ice", "Sand" del package "model"

3.3 Note di sviluppo

3.3.1 Zanetti Lorenzo

- Uso di lambda expressions
- Uso di Stream
- Uso di Optional
- Uso di java.awt.geom, libreria che gestisce geometricamente figure 2-dimensionali
- Uso di javax.swing.JLayeredPane, pannelli senza layout che gestiscono la sovrapposizioni di più elementi

3.3.2 Pedrini Fabio

- Uso di Optional
- Uso di lambda expressions
- Uso di java.awt.geom, libreria che gestisce geometricamente figure 2-dimensionali
- Uso di generici e di generici bounded

Considerando l'uso di generici bounded, vengono utilizzati in particolare nel metodo sortByValue() all'interno della classe NavigatorImpl. Questo metodo è stato personalizzato e selezionato tra un insieme di altri algoritmi presi dal sito Stack Overflow, non avendo trovato alternative efficaci in librerie esistenti.

3.3.3 Zanzi Alessandro

- Uso di listener
- Uso di ArrayList, List
- Uso di Optional
- Uso di javax.swing

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Zanetti Lorenzo

Mi ritengo soddisfatto del mio lavoro svolto per il progetto Jolf e dell'effettivo risultato finale che abbiamo prodotto. In particolare credo di aver svolto un buon lavoro nella progettazione iniziale dell'applicazione, nell'aver dato una base da cui partire al mio gruppo creando le prime classi (vuote), le prime interfacce e il menù iniziale (implementandolo) e nell'aver assistito gli altri membri quando necessario. Mi sembra di essere stato un po' la figura "leader" del team e di aver preso molte delle decisioni, sono contento di questo, ma allo stesso tempo avrei potuto dedicare più tempo a riguardare il codice scritto da me e a migliorarlo, infatti penso di dover migliorare su questo in futuro. Per quanto riguarda il futuro del progetto penso che, finito lo scopo didattico, al massimo implementerò qualche altro oggetto o mappa e lo condividerò con qualche amico o parente.

4.1.2 Pedrini Fabio

Secondo la mia opinione, il gruppo ha fatto un buon lavoro e ha superato le mie aspettative. Dato che la proposta di progetto era stata fatta un po' troppo tardi mi aspettavo un progetto di qualità molto inferiore rispetto a quello attuale e penso di aver dato un buon contributo all'interno del team. Ho avuto qualche difficoltà nella parte iniziale del progetto, ma sono riuscito a superare queste difficoltà grazie all'aiuto del team, in cui ci siamo aiutati in maniera reciproca. In generale, valuto positivamente il mio lavoro e sono soddisfatto di quello che ho prodotto, anche se è presente qualche aspetto che vorrei migliorare (come l'utilizzo di pattern o di librerie o la capacità di

trovare soluzioni più brevi per scrivere parti di codice). Penso che utilizzerò il progetto in futuro come "template" per altri progetti che possono avere caratteristiche simili a questo, magari aggiungendo nuove mappe o nuovi oggetti.

4.1.3 Zanzi Alessandro

Mi ritengo abbastanza soddisfatto del mio lavoro svolto per il progetto Jolf e di come si presenta l'applicazione. Penso sicuramente di essere l'elemento più indietro per quanto riguarda conoscenze e pratica con l'uso di java, dato che ancora devo sostenere l'esame pratico. Mi sono fatto guidare dai miei collaboratori chiedendo sempre un parere sulla mia parte di codice, cosa che avrei fatto in qualunque caso, e penso alla fine di essere riuscito ad arricchire le mie conoscenze tecniche. Questa visione "meno esperta" mi ha permesso però di poter valutare molti aspetti realizzativi del progetto facendo risaltare errori e miglirie apportabili, che rendessero più piacevole ed accurata l'esperienza di gioco. Per il futuro punto sicuramente a migliorare le mie capacità come programmatore java e nel migliorare la scrittura del codice. Per il progetto penso che lo userò come base di studio, ma anche per svago, impementando magari nuove funzionalità o aggiungendo mappe e ostacoli.

4.2 Difficoltà incontrate e commenti per i docenti

Zanzi Alessandro

Segnalo fra le problematiche riscontrate durante lo svolgimento del progetto, principalmente nella fase iniziale, alcuni problemi con git, in particolare nella possibilità di fare la git push e caricare le mie modifiche su GitHub.

Appendice A

Guida utente

All'avvio dell'applicazione, l'utente si trova di fronte alla home-page di Jolf, in cui potrà scegliere se iniziare subito una partita (tasto PLAY), controllare la classifica (tasto LEADERBOARD), esercitarsi con le mappe singole senza salvare il punteggio finale (tasto LEVELS), oppure accedere al menù di opzioni (OPTIONS) in cui potrà cambiare il colore della pallina o azzerare i punteggi nella classifica. Dal tasto PLAY si accede ad una finestra intermedia che richiede l'inserimento di un nome nella casella di testo e la scelta di un percorso cliccando il tasto corrispondente. Dal tasto LEVELS si accede ad una finestra intermedia analoga in cui servirà semplicemente cliccare il tasto della mappa che si vuole scegliere. Per lanciare la pallina basta trascinare il cursore partendo da un qualsiasi punto sullo schermo, tenendo premuto per poi rilasciare; la direzione e l'intensità del lancio sono date dall'indicatore rosso che appare sulla pallina.

Elementi della mappa

Le aree colorate di giallo e azzurro sono ostacoli che corrispondono rispettivamente a sabbia e ghiaccio; questi interagiscono con la pallina quando ci passa sopra, creando un attrito diverso da quello dell'erba e quindi diminuendo o aumentando la distanza percorsa. I rettangoli marroni rappresentano dei muri e i triangoli arancioni degli altri ostacoli fisici su cui la pallina rimbalzerà. L'obiettivo come descritto nell'introduzione è quello di "toccare" con la pallina due delle tre stelle presenti nella mappa e entrare nella buca (il cerchio bianco che si forma in corrispondenza della terza stella) nel minor numero di tiri possibile.

Appendice B

Esercitazioni di laboratorio

B.0.1 Zanetti Lorenzo

- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p101873`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p102842`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p104007`