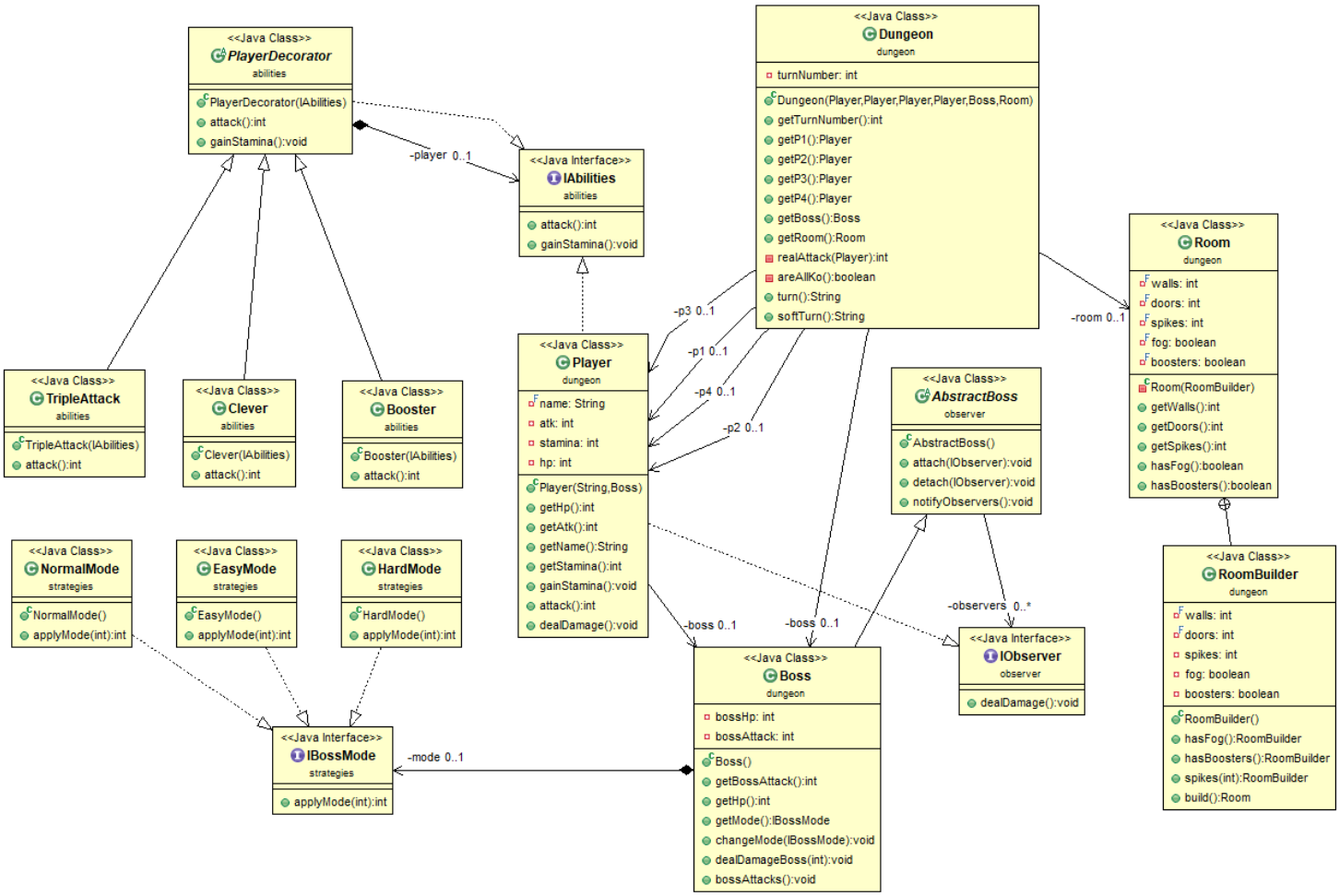
RELAZIONE PROGETTO METODOLOGIE DI PROGRAMMAZIONE:
Dungeon e minigame

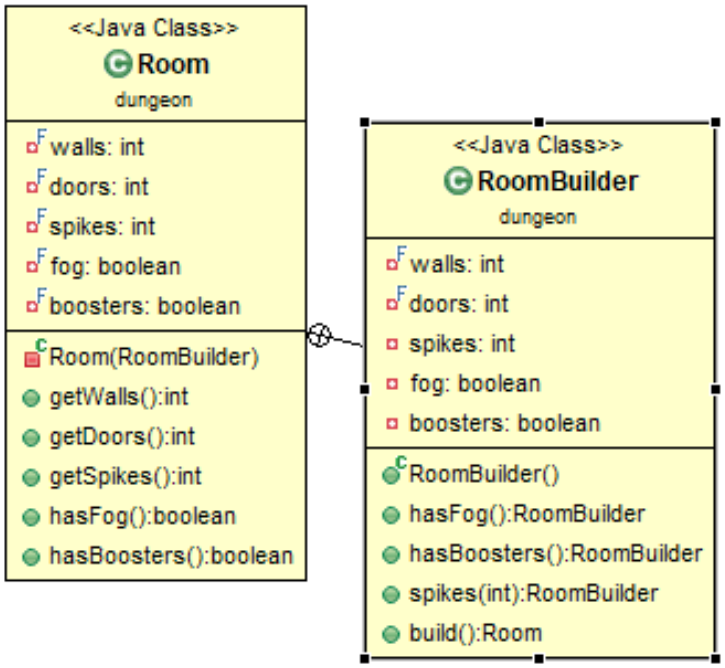
Il sistema implementato vuole simulare un minigame dove quattro personaggi sfidano un boss in un dungeon.
I personaggi sono definiti dalla classe "Player", il boss dalla classe "Boss" ed il tutto si svolge in una stanza definita dalla classe "Room".
Ogni Player ha un suo valore di attacco, dei punti vita e un valore di stamina, mentre il Boss possiede solamente valori di attacco e punti vita.
Il boss però può essere posto dinamicamente in tre modalità: facile, media e difficile, che varieranno il suo valore di attacco.
Ad ogni player possiamo assegnare varie abilità: "Booster", "Clever" e "Triple Attack", che ridefiniscono il metodo attack() di ogni player.
Quando il boss eseguirà un attacco attaccherà sempre contemporaneamente tutti i player.
I player e il boss si sfidano in una stanza che prevede dei parametri obbligatori: walls e doors che saranno sempre rispettivamente quattro e due, e dei parametri opzionali: fog, spikes e booster.
Inoltre se nella stanza sono presenti booster gli attacchi dei player avranno un potenziamento di 100.
Tutto è gestito dalla classe Dungeon, che possiede anche due turni tipo, e un parametro che tiene il conto di turni usati per portare i punti vita del boss oppure quelli di tutti i player a 0.



PATTERN USATI:

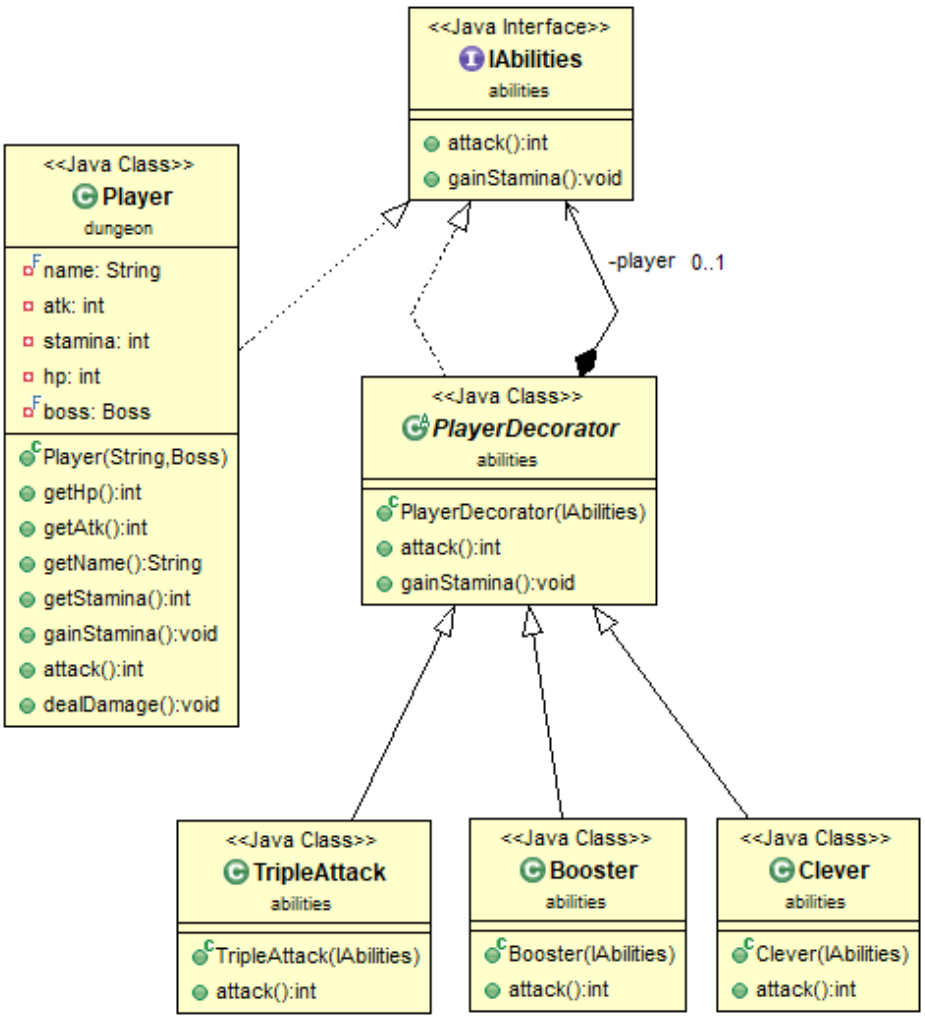
- Observer
- Strategy
- Decorator
- Builder (che fa uso di Fluent Interface)

CLASSE ROOM E PATTERN BUILDER



La classe Room permette di creare la stanza dove si svolgerà il minigame.
Ogni stanza dovrà possedere due parametri obbligatori "doors" e "walls", che rappresentano due porte e quattro mura, ed inoltre avrà 3 parametri opzionali: "spikes", "fog" e "boosters", in modo da creare stanze diverse tra loro.
La diniamicità nella creazione della stanza è resa possibile grazie all'applicazione del pattern Builder, infatti la classe Room contiene al suo interno una classe statica RoomBuilder, che ci permette di gestire la costruzione della stanza separandola dalla sua rappresentazione.
RoomBuilder, infatti, avrà solamente il compito di configurare gli oggetti di tipo Room, sostituendo il costruttore, che non ci avrebbe permesso di poter creare stanze diverse tra loro.
Il Builder si basa sull'utilizzo di un altro pattern, Fluent Interface, consentendoci di concatenare diversi metodi permettendoci di creare i vari tipi di campo.
In particolare questi metodi sono spikes(int) che gestisce il numero di punte decorative che avrà la stanza, hasFog() che definisce se nella stanza ci sarà nebbia, e hasBoosters() che ci indica se nella stanza sono presenti oggetti che potenziano l'attacco dei giocatori.
Infatti, se il valore boosters della stanza sarà true i quattro player riceveranno ognuno un boost di 100 punti ogni volta che eseguiranno un attacco.
Infine abbiamo il metodo build() che crea la stanza con tutte le caratteristiche scelte precedentemente.
I parametri della classe Room sono dichiarati tutti final, e quindi non è stato necessario implementare i setters, ma solo i vari getters.

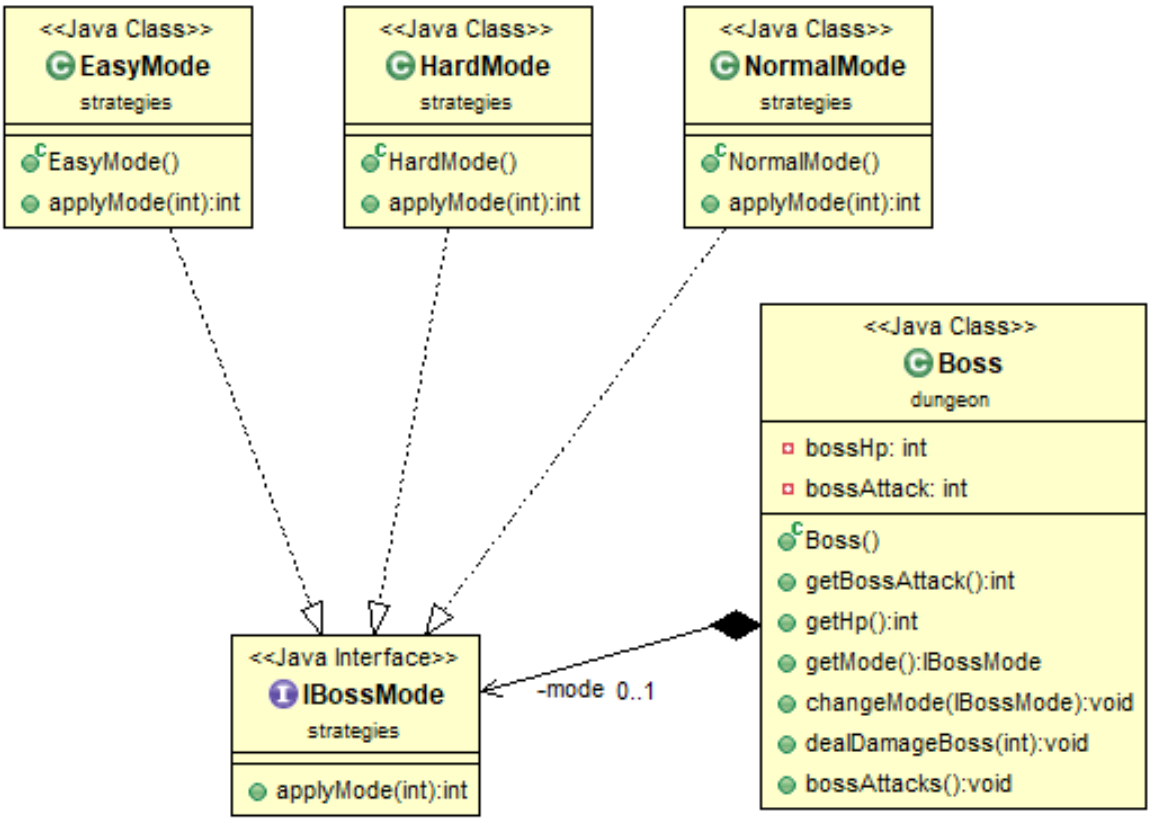
CLASSE PLAYER E PATTERN DECORATOR



La classe Player definisce i personaggi che dovranno sconfiggere il boss.
Ogni personaggio possiede un valore di attacco pari a 150, un valore di stamina uguale a 5000, degli hp settati a 1000, un nome e il riferimento ad un boss.
Possiede inoltre i getters di hp, atk, name, stamina. Inoltre boss e name sono dichiarati final.
Il metodo gainStamina() permette al player di recuperare punti stamina, che non possono però superare il valore iniziale di 5000, e ad ogni chiamata restituisce 300 punti, a meno che non siano maggiori di 4700, in quel caso la stamina verrà portata direttamente a 5000.
Il metodo dealDamage() permette al player di subire danni inflitti dal boss, e il suo funzionamento verrà descritto quando spiegherò il pattern Observer.
Infine il metodo attack() restituisce un valore int che rappresenterà il danno finale che il player infliggerà al boss: se gli hp o la stamina sono uguali a 0 il valore ritornato sarà 0, mentre negli altri casi il valore verrà calcolato controllando il valore della stamina: infatti ad ogni attacco essa viene diminuita di 300, e se l'attuale stamina risulta essere minore di 300 l'attacco sarà più leggero restituendo il valore attuale della stamina, e poi la stamina viene settata a 0.
Ogni player viene decorato sfruttando il pattern decorator, e questo permette ad ognuno di essi di avere 3 abilità: triple attack, booster e clever.
Ognuna di esse ridefinisce il metodo attack richiamandolo con il comando super:

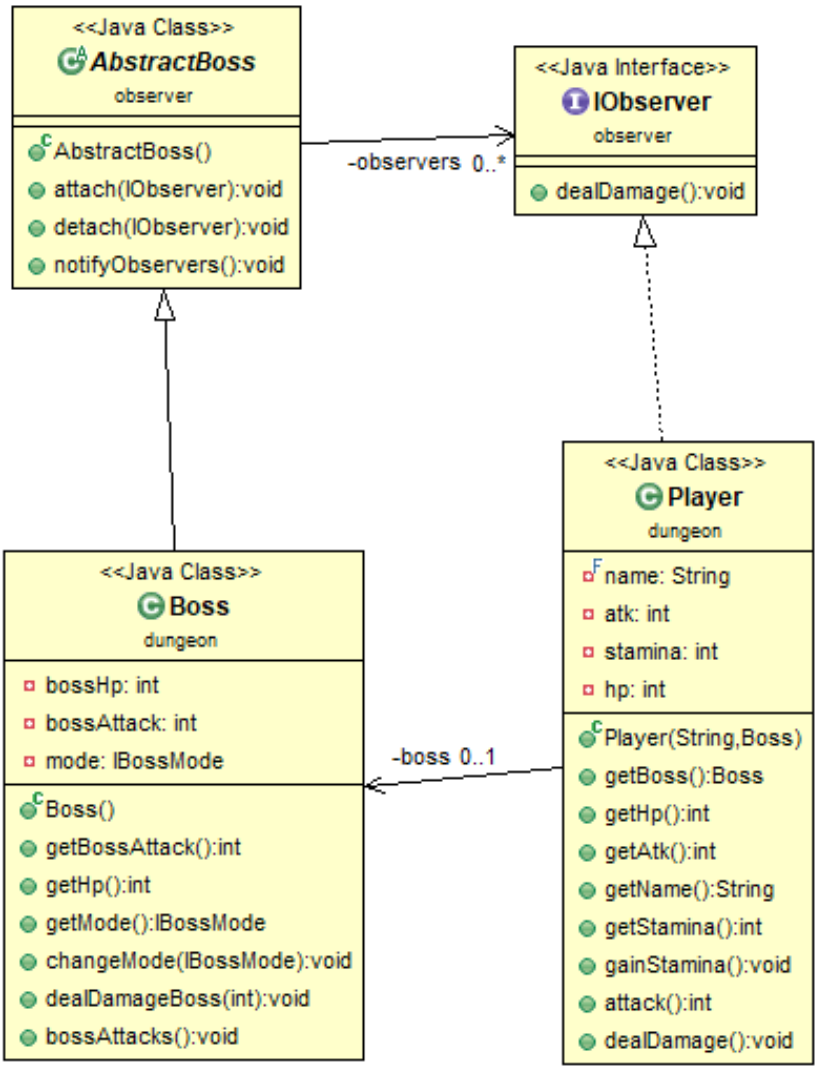
- TripleAttack che al posto di restituire attack() una volta lo restituisce tre volte
- Booster che potenzia l'attacco di 1000 punti
- Clever che prima di attaccare invoca il metodo gainStamina(), permettendo al player di non fare azzerare mai i punti stamina.

CLASSE BOSS E PATTERN STRATEGY



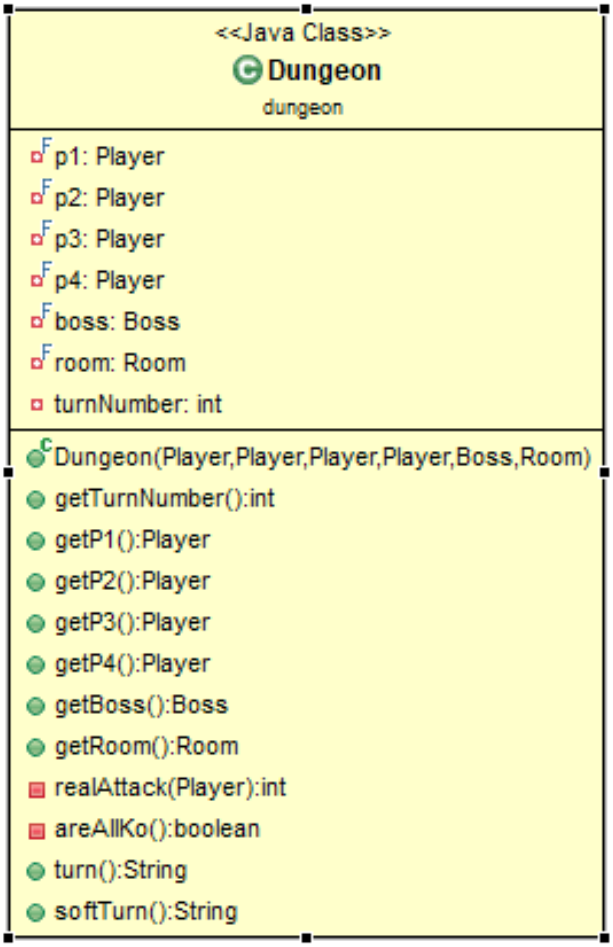
La classe Boss definisce il boss che dovrà essere sconfitto dai quattro player.
Ogni boss possiede due parametri: bossHp che indica i punti vita del boss, inizialmente posti a 10000, e bossAtk che indica il valore di attacco del boss, settato a 100.
Inoltre il boss può essere settato su tre diverse modalità: easy mode, normal mode e hard mode.
Ogni modalità modifica il parametro di attacco del boss, infatti il metodo getBossAttack() contiene della logica.
Le varie modalità possono essere cambiate dinamicamente sfruttando il pattern strategy: infatti con il metodo changeMode() posso cambiare la modalità corrente del boss, che inizialmente è normal.
Finchè la modalità è settata su normal l'attacco del boss non subirà alcuna variazione, se la modalità scelta sarà hard il boss infliggerà il doppio del danno ai player, mentre se la modalità sarà easy il boss infliggerà la metà del danno.
L'attacco finale del boss sarà modificato quindi dal metodo applyMode(), diverso per ognuna delle tre strategie implementate.
Il metodo dealDamageBoss(int) riceve in ingresso un valore int, ovvero il danno inflitto dai player, e diminuisce i punti vita del boss del valore inserito, fino a farli scendere a 0.
I punti vita non possono ovviamente scendere sotto lo 0.
Il boss può inoltre attaccare i player, grazie al metodo bossAttacks(), e all'utilizzo del pattern observer.

PATTERN OBSERVER



Il boss può attaccare i player grazie al metodo bossAttacks(), e quando lo farà attaccherà tutti i player, utilizzando il suo attacco variato dalla modalità scelta.
Quando viene chiamato il metodo bossAttacks() vengono notificati gli observers, precedentemente attaccati grazie al metodo attach(IObserver), sfruttando il metodo notifyObservers().
Gli observer possono anche essere staccati col metodo detach(IObserver).
AbstractBoss è astratta anche se contiene metodi non astratti per evitare che la si istanzi.
IObserver è l'interfaccia dei ConcreteObservers di tipo Player, e contiene il metodo dealDamage(), il metodo update che verrà ridefinito dai giocatori.
Quando i player sono avvisati richiameranno il metodo dealDamage(), che diminuisce i loro hp del valore restituito dal metodo getBossAttack(), e se gli hp correnti sono minori di quel valore li diminuisce a zero, in modo che gli hp non siano mai negativi.

CLASSE DUNGEON



La classe Dungeon costruisce l'oggetto finale dove si svolgerà il minigame.
Deve essere istanziata con 4 player un boss e una room.
Quando viene costruito i player vengono automaticamente attaccati al boss usando attach(IObserver), e il valore turnNumber che rappresenta il numero di turni passati dall'inizio viene settato a 0.
Tutti i campi sono dichiarati come final.
Successivamente abbiamo due metodi privati: realAttack(Player) che applica il boost di 100 punti agli attacchi dei player se nella stanza sono presenti i booster, e areAllKo() che restituisce true solo se tutti i player hanno i punti vita ridotti a zero.
Questi due metodi verranno sfruttati dai metodi turn, che gestiscono i vari turni dei minigame. I turni restituiranno delle stringhe che descrivono cosa è avvenuto.
Sono anche stati implementati due turni tipo:

- turn(): inizialmente controlla che tutti i player siano in vita, successivamente controlla se il boss è già stato sconfitto e infine esegue il vero e proprio turno: prima attaccano i quattro player, poi attacca il boss, successivamente viene incrementato il numero dei turni e infine viene restituita la stringa descrittiva del turno.
- softTurn(): inizialmente vengono controllate le stesse condizioni descritte in turn(), e successivamente invece che tutti e quattro attaccano solamente i primi due player. Tutto il resto del turno resta invariato, quindi il boss attacca, i turni vengono aumentati e viene restituita la stringa descrittiva.

STRATEGIE DI TESTING

Per testare le varie parti del progetto ho deciso di suddividere i test in modo da poter testare ognuna delle classi più importanti del sistema, quindi sono suddivisi in:

- RoomTest
- PlayerTest
- BossTest
- DungeonTest

RoomTest: In RoomTest viene testata interamente la classe Room, e quindi il pattern Builder.
Inizialmente ho creato due campi: r2 che prevede l'utilizzo di tutti i parametri possibili e
r1, che prevede solamente i due parametri necessari walls e doors.
Successivamente ho testato con i vari metodi get che entrambi le room avessero tutti i parametri assegnati, ed anche che r1 non contenesse spikes, fog e boosters.

PlayerTest: In PlayerTest, oltre a testare il corretto funzionamento di Decorator ho testato anche i metodi gainStamina e attack, sfruttando degli assertEquals.
I tre vari decorator sono stati testati su tre player diversi, controllando che i nuovi comandi attack si comportino correttamente.
Infine ho anche testato che gli hp non scendessero sotto lo 0 anche se il boss infliggesse un quantitativo di danni superiore ai punti vita rimanenti ai player.

BossTest: In BossTest inizialmente viene testato che gli hp del boss non possano diventare negativi come in PlayerTest, e successivamente vengono testati i pattern Observer e Strategy:
Per testare Observer vengono attaccati al boss due player, viene chiamato il metodo bossAttacks() e poi con degli assertEquals vengono controllati gli hp dei player. Successivamente viene anche controllato che detach(IObserver) funzioni.
Per testare Strategy invece inizialmente viene testato che il boss sia di default settato su normal, e successivamente con degli assertEquals viene controllato che l'attacco del boss effettivamente cambi.

DungeonTest: In DungeonTest viene controllato il corretto funzionamento di dungeon, testando i turni e con un esempio di sconfitta:
Dopo aver creato l'oggetto dungeon vengono controllate le stringhe descrittive e con i vari get che gli hp del boss e dei player effettivamente vengano diminuiti, sia chiamando turn() che softTurn().
Viene anche controllato che se il boss è stato sconfitto venga restituita la stringa corretta e che non venga fatto più niente.
Infine abbiamo un test per una possibile sconfitta, dove viene controllato che venga restituita la giusta stringa e che non venga fatto più niente.