Relazione del Progetto relativo l'insegnamento di Programmazione e Modellazione a Oggetti per la sessione Autunnale 2021/2022

Relatori:

Attarantato Kevin, Matricola 282785

Roselli Giorgia, Matricola 282724

Docente:

Prof. Delpriori Saverio

Specifica del problema

Lo scopo del progetto è quello di realizzare un'applicazione desktop, su singola schermata, per un gioco chiamato "Monster Hunter": un classico gioco di ruolo, che avrà come protagonista un eroe, una storia formata da un susseguirsi di missioni, ed una serie di combattimenti a turni contro dei nemici che si incontreranno lungo il cammino.

1. Il giocatore inizierà in un luogo prestabilito, o nel caso si fosse salvata la partita dell'ultimo luogo visitato;
2. Da lì potrà muoversi in diverse location;
3. Nell'avanzare nei diversi luoghi, potrebbe ricevere una missione;
4. Per completare una missione sarà necessario consegnare diversi oggetti al mandante della missione;
5. Per collezionare i diversi oggetti richiesti, il giocatore dovrà muoversi nell'ambiente di gioco sconfiggendo diversi nemici per ottenere gli oggetti necessari alla riuscita della missione;
6. Il giocatore affronterà tali nemici attraverso l'uso di armi, anche se ad ogni utilizzo di un qualsiasi oggetto verrà attaccato dal mostro corrente;
7. Le armi saranno ottenibili sconfiggendo determinati nemici;
8. Il giocatore potrà anche curarsi attraverso l'utilizzo di pozioni curative, ottenibili anch'esse sconfiggendo determinati nemici;
9. Il giocatore potrò in qualsiasi momento salvare e visionare la mappa;
10. In caso si fosse salvata la partita al prossimo avvio il gioco ripartirà da dove il giocatore aveva lasciato;
11. Il gioco terminerà con la sconfitta del boss finale lasciando al giocatore la possibilità di continuare a giocare illimitatamente.

Studio del problema

I punti critici nella progettazione del suddetto software sono stati i seguenti:

1. Voler dividere la parte riguardante la logica del programma, dalla parte grafica del software per rendere il suddetto più facile da leggere, da testare e da debuggare qual ora si verificassero problemi durante la scrittura;
2. Creare, nella parte logica, una classe che contenesse le varie interconnessione fra le diverse classi che creano i diversi elementi del programma (es. pozioni, armi, mostri ecc.);
3. Interfaccia grafica che permette al giocatore di poter giocare utilizzando i vari elementi citati nel precedente punto;
4. Creare un file che potesse ospitare le informazioni del giocatore cosi da poterle ricaricare al seguente avvio dell'applicazione previo salvataggio da parte del giocatore.

Da cui sono derivate le seguenti scelte:

1. Per risolvere questo problema, la scelta che si è voluta fare è stata quella di andare a creare un progetto con la soluzione (applicazione) divisa in due parti dove:
    - La prima componente è stata fatta di tipo *'Windows Forms Application'*, cosicché contenga il vero e proprio form del progetto e quindi l'effettiva parte grafica che verrà visualizzata a schermo;
    - La seconda componente è stata fatta, invece, di tipo *'Class Library'*, che quindi andrà a costruire una libreria contenente i vari elementi che andranno poi a costruire il programma;

    Fatto questo quello che si è fatto è andare a collegare le due parti, andando ad aggiungere alla lista delle referenze del progetto di tipo *'Windows Forms Application'* il secondo progetto creato.

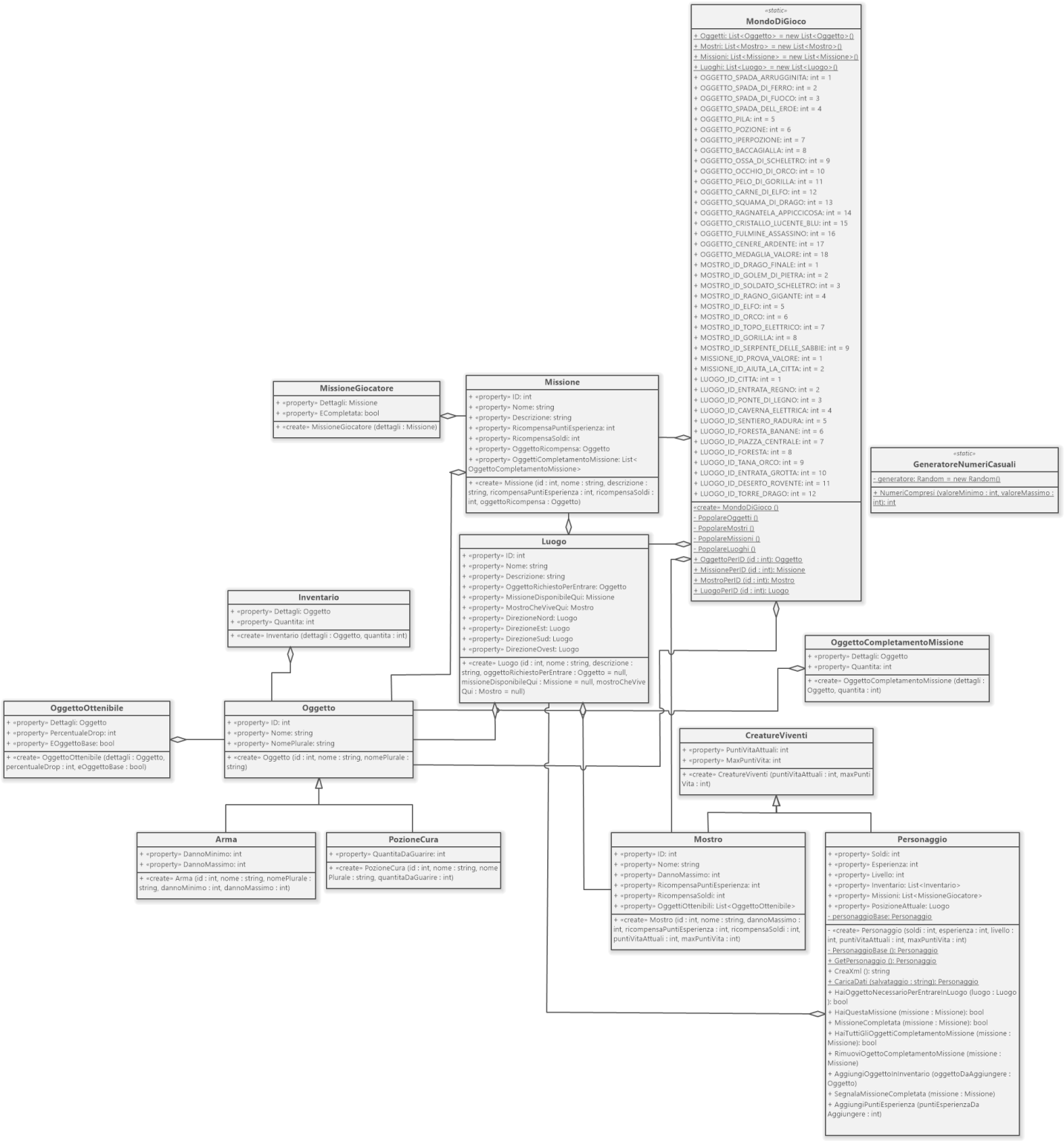
2. Per raggruppare tutte le classi che dovranno interagire fra di loro, andando quindi poi a popolare il mondo di gioco, si è pensato di costruire una classe ad hoc che conterrà, quindi, tutte le connessioni necessarie per andare a costruire e popolare il gioco. Oltre le classi che faranno riferimento ai singoli elementi di gioco come pozioni, armi, missione ecc. c'è ne sarà una, chiamata *'MondoDiGioco'*, dove si andranno ad istanziare i vari elementi (missioni, oggetti, mostri ecc.) e caricati in apposite liste. Questa classe, inoltre, sarà statica (quindi presenterà la possibilità di accedervi solo attraverso il nome della classe stessa) dalla quale sarà possibile ottenere, attraverso 4 metodi pubblici, i vari elementi dalle relative liste, restituendo quindi l'elemento stesso all'invocante.

3. L’interfaccia grafica è costituita da due schermate distinte, una principale ed una secondaria con la possibilità di accedervi dalla principale durante l’esecuzione del software. In particolare la prima permetterà al giocatore di poter intraprendere tutte le scelte che lo condurranno, poi, a terminare l’avventura quindi attaccare un mostro, spostarsi fra le diverse regioni, acquisire missioni. La seconda interfaccia, invece, sarà accessibile da quella principale come detto poc’anzi, e conterrà la mappa di gioco, quindi le varie immagini che lo compongono con raffigurate delle frecce rosse per far capire al giocatore le eventuali direzioni percorribili, oltre che ad un breve testo per la spiegazione di quest’ultimo concetto ed un pulsante per la chiusura dell’interfaccia stessa.
4. Per risolvere quest’ultimo problema abbiamo pensato inizialmente a diverse soluzioni, che potessero essere coerenti con quello che dovevamo fare, tra cui l’utilizzo di un DB in mysql, un semplice file di testo oppure un file XML. Tra queste scelte si è optato per l’utilizzo di un file XML (eXtensible Markup Language) il quale non necessiterà l’installazione di nessun programma o libreria extra e sarà molto semplice da leggere in quanto il file conterrà TAG human readable creati appositamente ed i dati di cui avremo bisogno. La creazione, per quanto in prima battuta sia stata fatta manualmente, sarà fatta da riga di codice all’interno della classe *‘Personaggio’* per una coerenza con i dati che dovrà andare ad ospitare. Anche il metodo per il caricamento dei dati è stato inserito nella classe citata pocanzi per lo stesso motivo. Parlando di quest’ultimo metodo si è scelto, in caso fosse stato creato un file di salvataggio, di caricarlo automaticamente all’avvio seguente dell’applicazione, previo salvataggio da parte del giocatore.

Scelte architetturali

Si è scelto, per una maggior chiarezza, di suddividere il diagramma delle classi in due componenti che corrispondono alle due parti sopra citate, quindi relativamente alla parte logica del programma, ed una alla parte grafica:

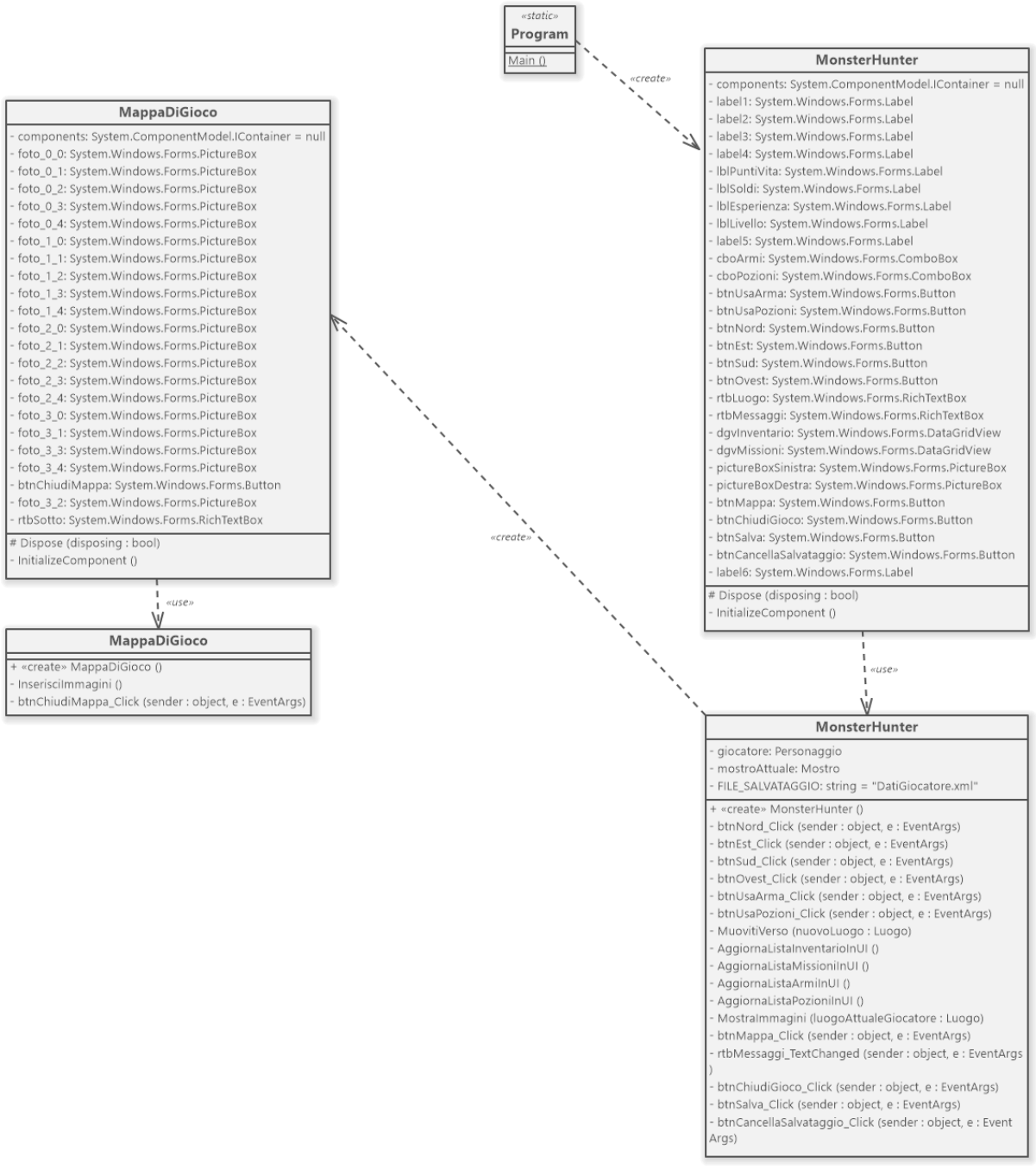
- **Parte Logica**:



Nel diagramma sopra riportato vengono utilizzate le seguenti relazioni:

- Relazioni di generalizzazioni, in quanto le classi *'Arma'* e *'PozioneCura'* estendono la classe *'oggetto'*, mentre le classi *'Mostro'* e *'Personaggio'* estendono la classe *'CreatureViventi'*;
- Relazioni di aggregazione, in quanto rappresenta la relazione "fa parte di" e che si presenta quando una classe è un contenitore di altre classi, ma le classi contenute non dipendono fortemente dal contenitore, quindi se venisse distrutto il contenitore, esse continuerebbero a vivere.

- **Parte Grafica**:



In quest'ultimo diagramma, invece, viene utilizzata la seguente relazione:

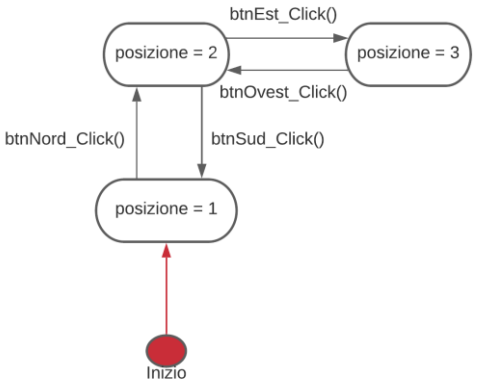
- Relazione di dipendenza, in quanto la main, quindi il punto di partenza dell'applicazione, crea un oggetto di tipo *MonsterHunter* il quale farà riferimento al suo codice. Tale codice andrà poi a creare il form *'MappaDiGioco'* che a sua volte userà il suo codice.
  Questo genere di relazione è utile per definire una relazione che esiste fra due elementi quando cambiando la definizione di un elemento, viene cambiato anche l'atro.

Design Pattern utilizzati:

1. Pattern Creazionali:
   - **Singleton**: Questo pattern è stato implementato per la classe *'Personaggio'*, in quanto solo un'istanza deve essere istanziabile a tempo di esecuzione. Questa classe, perciò, presenta un costruttore privato, un attributo privato statico che memorizza l'istanza specifica ed un metodo pubblico per l'acquisizione dell'istanza appena creata, la quale restituirà, quindi, tale unica istanza;
   - **Factory method**: Questo pattern utilizza i "factory methods" per andare a creare oggetti senza specificare l'esatto oggetto. Questo viene fatto creando oggetti richiamando un metodo piuttosto che richiamando il costruttore. 'Factory' tradotto in italiano è fabbrica, infatti questo pattern produce istanze, un po' come una fabbrica produce prodotti. Questo pattern è stato implementato sempre nella classe *'Personaggio'* in quanto il costruttore di questa classe è privato (questo per implementazione del pattern singleton) e quindi risulta impossibile istanziare un oggetto di questo tipo se non attraverso il metodo *'GetPersonaggio'*. Qual ora, però, il giocatore avesse salvato una partita bisognerebbe andare a creare un personaggio sulla base dei dati passati, questa operazione è svolta dal metodo *'CaricaDati'* che si occuperà di andare a creare un'istanza di *'Personaggio'* per noi sulla base dei dati passati del giocatore. Qual ora ci fossero problemi con questo metodo, comunque, verrebbe istanziato un personaggio base.
     Questo metodo rispetta anche il pattern singleton in quanto sempre una ed una soltanto istanza di questa classe verrà creata è sarà sempre la stessa ad essere richiamato quando si invocherà *'GetPersonaggio'*.
2. Pattern strutturali:
   - **Flyweights**: Questo pattern permette, di separare la parte variabile di una classe dalla parte che può essere riutilizzata, in modo tale da condividere quest'ultima fra differenti istanze, e di ridurre al minimo l'utilizzo di memoria ove possibile. Questo pattern è stato implementato con la classe

'GeneratoreNumeriCasuali', in quanto, il risultato, sarà necessario nella classe *'MonsterHunter.cs'* per ottenere dei valori casuali compresi fra il valore minimo ed il valore massimo dell'arma equipaggiata. Per ridurre la quantità di variabili (e quindi di memoria) utilizzate si è voluta creare questa classe immutabile che potrà essere riutilizzata fra le diverse classi senza creare ogni volta variabili volte ad ospitare questi valori.

3. Pattern comportamentali:
   - **State**: Possiamo schematizzare la possibilità di accedere fra i diversi luoghi creati cambiando uno stato interno della classe *'Personaggio'*. Possiamo dunque modellare la seguente macchina a stati finiti sulla quale progettare uno state pattern:



Lo stato iniziale sarà il punto di partenza del giocatore, che corrisponderà ad un determinato luogo, e si potrà cambiare quest'ultimo tramite i bottoni *'Nord'*, *'Sud'*, *'Est'*, *'Ovest'* del form. Tale pattern, quindi, è costituito dall'interfaccia di gioco ed i 4 possibili stati che la implementano.

Considerazioni finali:

Per quanto riguarda il pattern singleton è buona norma, quando si crea un metodo per richiamare l'unica istanza che si è creata, mettere un'istruzione di sincronizzazione prima dell'istruzione di selezione in questo modo:

```
public static Logger GetLogger()
{
    if(_logger == null)
    {
        lock (_syncLock)
        {
            if(_logger == null)
            {
                _logger = new Logger();
            }
        }
    }

    return _logger;
}
```

questo nel codice di 'MonsterHunter' non è stato volutamente fatto in quanto non succederà mai che più oggetti richiameranno quel metodo e quindi esponendolo a rischi di sincronizzazione.

Infine, segnalo che è stato utilizzato un quarto pattern chiamato **MVVM pattern** (Model-View-ViewModel) il quale è simile al MVP (Model-View-Presenter) e MVC (Model-View-Controller) e facilita la separazione dei dati (model) dall'interfaccia grafica (view) in maniera tale che l'interfaccia non influenzi la gestione dei dati e che essi possano essere organizzati senza dover apportare cambiamenti all'interfaccia utente.

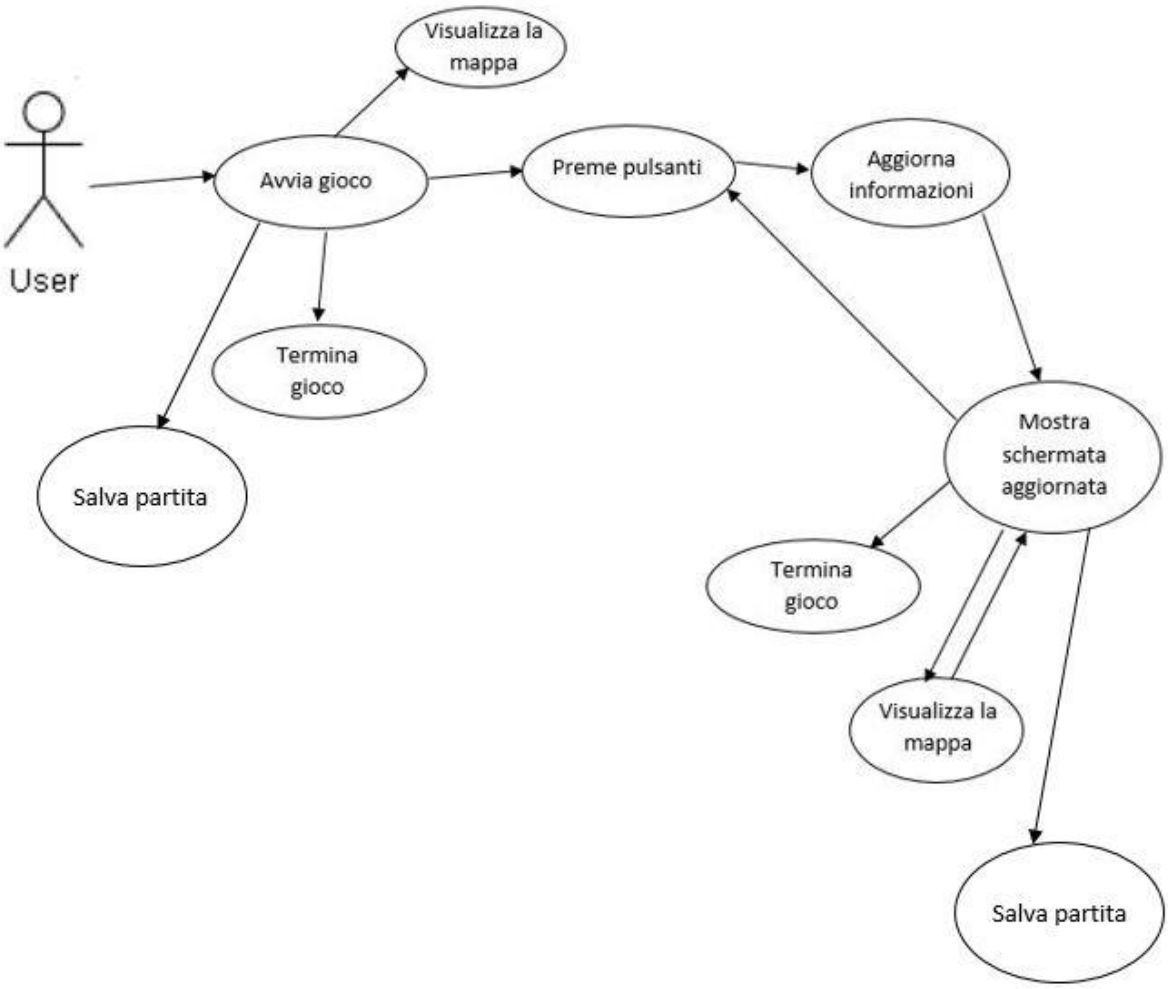
Documentazione sull'utilizzo

- Il software è stato realizzato usando la versione 'Visual Studio 2019' gratuita aggiornato alla versione 16.10.4 su un Sistema Operativo Windows 10 pro con architettura a 64 bit;
- L'applicativo è stato progettato tramite il framework grafico Windows Forms;
- Per eseguire l'applicazione bisognerà entrare dentro la cartella *'MonsterHunter'* e fare doppio click sul file con estensione .sln e cliccare poi 'compila';
- In alterativa è possibile entrare nella cartella chiamata *'Eseguibile'* presente nella repository e fare doppio click sul .exe;
- Non sono necessari particolari parametri per l'esecuzione o la compilazione della soluzione.

Contributo degli studenti

Abbiamo, per quanto riguarda la creazione del codice e la stesura della relazione, sempre voluto cooperare affinché il lavoro venisse svolto nel miglior modo possibile perciò non c'è stata una vera e propria divisione del carico di lavoro o delle varie componenti; Tuttavia è possibile dire che lo studente Attarantato si sia concentrato maggiormente sulla front-end e quindi sulla scrittura del codice riguardante l'interfaccia grafica e della classe Personaggio, mentre la studentessa Roselli si è concentrata di più sul back-end quindi sulla scrittura dell'intera parte logica del programma, a parte la classe 'Personaggio' la quale aveva solo creato ed inserito parte degli attributi finali.
Per quanto riguarda la stesura della relazione si è collaborato sulla scrittura di ogni paragrafo e quindi risulta impossibile fare una vera e propria divisione del carico di lavoro.

Use cases con UML



L'utente dopo aver avviato il programma, visualizzerà il form che gli darà la possibilità di interagire con il medesimo, quindi potrà muoversi, o nel caso ci fossero mostri nel luogo in cui si trova, potrà attaccare e curarsi, tramite i relativi bottoni di interazione. Nell'interfaccia verranno aggiornate le informazioni quando l'utente sconfigge un mostro (vengono aggiornati i parametri in alto a destra oltre ad una stringa di testo nel box centrale) e quando l'utente si sposta da un luogo all'altro (vengono aggiornati foto descrittiva, testo di descrizione ecc.). Quando la schermata sarà aggiornata l'utente potrà re-intraprendere una delle scelte iniziali, quindi o spostarsi nel gioco oppure combattere un mostro qual ora ce ne fosse uno oppure visualizzare la mappa, tramite relativo bottone. La mappa di gioco sarà visualizzabile in qualsiasi momento. Il gioco è stato volutamente creato affinché non finisca mai, ma termini solo qual ora il giocatore premesse il bottone *'chiudi'*.