Relazione per "Geometry Survival"

Sergio Dobrianskiy
Pierangelo Motta
Thomas Testa
Matteo Trezza

10 giugno 2023

Indice

Capitolo 1

Obbiettivo

L'obbiettivo è di realizzare un videogioco ispirato a "Vampire Survivors" (Steampowered.com), un reverse bullet hell 2D con visuale dall'alto centrata sul giocatore. Il giocatore dovrà cercare di sopravvivere il più a lungo possibile a ondate di mostri che gli appariranno attorno. Le armi in possesso del giocatore verranno attivate in automatico durante la partita e si potranno potenziare salendo di livello.

1.1 Requisiti

Funzionalità minime ritenute obbligatorie:

- Movimento player (wasd o frecce direzionali) e nemici in campo
- Gestione delle hitbox e collisioni
- Creazione e gestione di oggetti e nemici con caratteristiche diverse
- Almeno 3 armi e 2 power up a disposizione dell'utente
- Gestione interfaccia grafica del giocatore
- Gestione livelli/statistiche giocatore

Funzionalità opzionali:

- Salvataggio e visualizzazione di una leaderboard che tiene traccia dei risultati degli utenti (classifica giocatori)

- Musica ed effetti sonori
- Più personaggi selezionabili
- Difficoltà multiple
- Aggiunta ostacoli statici sulla mappa
- Movimento con mouse

Challenge:

- Individuare, apprendere ed utilizzare correttamente dei pattern di progettazione adatti ai diversi obiettivi progettuali
- Gestione efficiente di un elevato numero di nemici presenti contemporaneamente sullo schermo
- Gestione degli sprite

Suddivisione del lavoro:

Sergio Dobrianskiy:

- Armi e potenziamenti
- Gestione delle collisioni tra entità

Pierangelo Motta:

- Creazione e spawn tipi di mostri diversi
- Gestione movimenti ed uccisione mostri e drop esperienza per power up armi player

Thomas Testa:

- Gestione del personaggio principale in ogni suo aspetto (incluso movimenti in qualsiasi direzione della mappa)
- Mappa e gestione dello sliding grafico della telecamera e movimento

Matteo Trezza:

- Gestione del lifecycle del gioco (inizializzazione gioco, inizio - fine partita, pausa)
- Grafica e sprite

1.2 Analisi e modello del dominio

Il giocatore verrà posizionato su mappa predefinita e non generato in modo casuale e dovrà essere in grado di muoversi nelle 4 dimensioni di un mondo 2d. La mappa dovrà avere dei bordi per impedire al giocatore di uscirne.

Attorno al giocatore appariranno dei mostri che tenteranno di raggiungerlo, se ci riusciranno inizieranno a causare danno. I mostri dovranno essere in grado di trovare il giocatore e cercare di inseguirlo e raggiungerlo.

Per difendersi dai mostri il giocatore sbloccherà delle armi che potranno essere potenziate. Ciascuna arma dovrà avere una sua meccanica, ad esempio sparare al nemico più vicino, far roteare i colpi attorno al giocatore, ecc.

Alla morte dei nemici dovranno apparire sul livello delle gemme che il giocatore dovrà poter raccogliere per guadagnare esperienza o vita. In caso sfortunato, alla morte di un nemico potrebbe apparire un nuovo mostro.

Si dovrà trovare il giusto numero di entità su schermo e un metodo efficace per controllarle per non compromettere le prestazioni del gioco.

Capitolo 2

Design

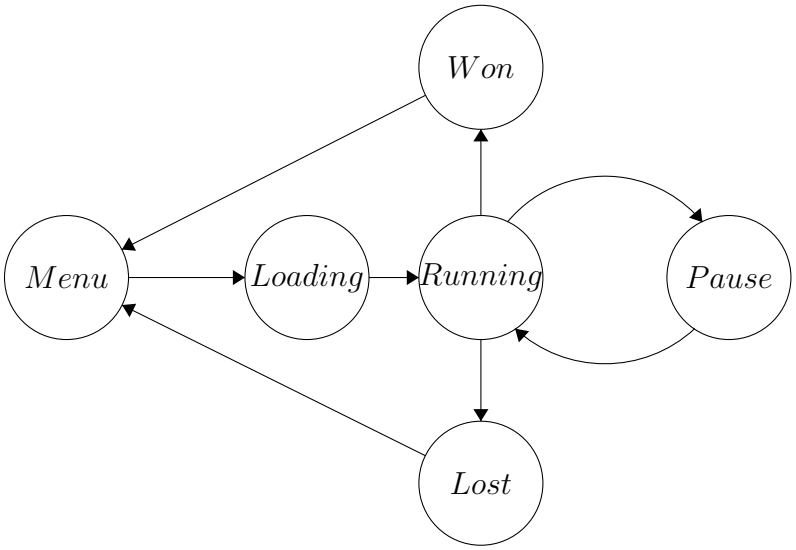
2.1 Architettura

TODO: Immgaine MVC

Abbiamo scelto di realizzare il gioco seguendo il pattern di programmazione MVC. ...

2.2 Design dettagliato

2.2.1 Finite State Machine/State pattern



Si parte da una visione architetturale, il cui scopo è informare il lettore di quale sia il funzionamento dell'applicativo realizzato ad alto livello. In particolare, è necessario descrivere accuratamente in che modo i componenti principali del sistema si coordinano fra loro. A seguire, si dettagliano alcune parti del design, quelle maggiormente rilevanti al fine di chiarificare la logica con cui sono stati affrontati i principali aspetti dell'applicazione.

2.3 Architettura

Questa sezione spiega come le componenti principali del software interagiscono fra loro. In particolare, qui va spiegato **se** e **come** è stato utilizzato il pattern architetturale model-view-controller (e/o alcune sue declinazioni specifiche, come entity-control-boundary).

Se non è stato utilizzato MVC, va spiegata in maniera molto accurata l'architettura scelta, giustificandola in modo appropriato.

Se è stato scelto MVC, vanno identificate con precisione le interfacce e classi che rappresentano i punti d'ingresso per modello, view, e controller. Raccomandiamo di sfruttare la definizione del dominio fatta in fase di analisi per capire quale sia l'entry point del model, e di non realizzare un'unica macro-interfaccia che, spesso, finisce con l'essere il prodromo ad una "God class". Consigliamo anche di separare bene controller e model, facendo attenzione a non includere nel secondo strategie d'uso che appartengono al primo.

In questa sezione vanno descritte, per ciascun componente architetturale che ruoli ricopre (due o tre ruoli al massimo), ed in che modo interagisce (ossia, scambia informazioni) con gli altri componenti dell'architettura. Raccomandiamo di porre particolare attenzione al design dell'interazione fra view e controller: se ben progettato, sostituire in blocco la view non dovrebbe causare alcuna modifica nel controller (tantomeno nel model).

Elementi positivi

- Si mostrano pochi, mirati schemi UML dai quali si deduce con chiarezza quali sono le parti principali del software e come interagiscono fra loro.

- Si mette in evidenza se e come il pattern architetturale model-view-controller è stato applicato, anche con l'uso di un UML che mostri le interfacce principali ed i rapporti fra loro.
- Si discute se sia semplice o meno, con l'architettura scelta, sostituire in blocco la view: in un MVC ben fatto, controller e modello non dovrebbero in alcun modo cambiare se si transitasse da una libreria grafica ad un'altra (ad esempio, da Swing a JavaFX, o viceversa).

Elementi negativi

- L'architettura è fatta in modo che sia impossibile riusare il modello per un software diverso che affronta lo stesso problema.
- L'architettura è tale che l'aggiunta di una funzionalità sul controller impatta pesantemente su view e/o modello.
- L'architettura è tale che la sostituzione in blocco della view impatta sul controller o, peggio ancora, sul modello.
- Si presentano UML caotici, difficili da leggere.
- Si presentano UML in cui sono mostrati elementi di dettaglio non appartenenti all'architettura, ad esempio includenti campi o con metodi che non interessano la parte di interazione fra le componenti principali del software.
- Si presentano schemi UML con classi (nel senso UML del termine) che "galleggiano" nello schema, non connesse, ossia senza relazioni con il resto degli elementi inseriti.
- Si presentano elementi di design di dettaglio, ad esempio tutte le classi e interfacce del modello o della view.
- Si discutono aspetti implementativi, ad esempio eventuali librerie usate oppure dettagli di codice.

Esempio

L'architettura di GLaDOS segue il pattern architetturale MVC. Più nello specifico, a livello architetturale, si è scelto di utilizzare MVC in forma "ECB", ossia "entity-control-boundary"[1]. GLaDOS implementa l'interfaccia AI, ed

[1]Si fa presente che il pattern ECB effettivamente esiste in letteratura come "istanza" di MVC, e chi volesse può utilizzarlo come reificazione di MVC.

è il controller del sistema. Essendo una intelligenza artificiale, è una classe attiva. GLaDOS accetta la registrazione di Input ed Output, che fanno parte della "view" di MVC, e sono il "boundary" di ECB. Gli Input rappresentano delle nuove informazioni che vengono fornite all'IA, ad esempio delle modifiche nel valore di un sensore, oppure un comando da parte dell'operatore. Questi input infatti forniscono eventi. Ottenere un evento è un'operazione bloccante: chi la esegue resta in attesa di un effettivo evento. Di fatto, quindi, GLaDOS si configura come entità *reattiva*. Ogni volta che c'è un cambio alla situazione del soggetto, GLaDOS notifica i suoi Output, informandoli su quale sia la situazione corrente. Conseguentemente, GLaDOS è un "observable" per Output.

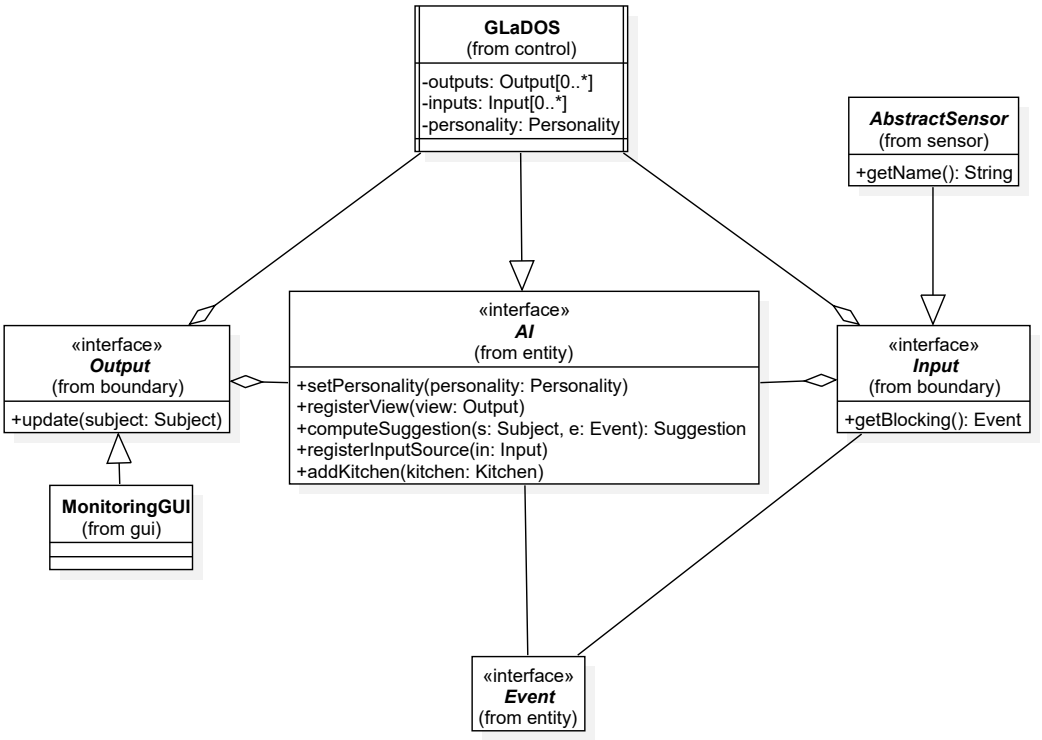


Figura 2.1: Schema UML architetturale di GLaDOS. L'interfaccia `GLaDOS` è il controller del sistema, mentre `Input` ed `Output` sono le interfacce che mappano la view (o, più correttamente in questo specifico esempio, il boundary). Un'eventuale interfaccia grafica interattiva dovrà implementarle entrambe.

Con questa architettura, possono essere aggiunti un numero arbitrario di input ed output all'intelligenza artificiale. Ovviamente, mentre l'aggiunta di output è semplice e non richiede alcuna modifica all'IA, la presenza di nuovi tipi di evento richiede invece in potenza aggiunte o rifiniture a GLaDOS.

Questo è dovuto al fatto che nuovi Input rappresentano di fatto nuovi elementi della business logic, la cui alterazione od espansione inevitabilmente impatta il controller del progetto.

In Figura 2.1 è esemplificato il diagramma UML architetturale.

2.4 Design dettagliato

Sergio Dobrianskiy:

1. Factory Pattern: WeaponFactory

    - Problema: creare un'istanza delle armi presenti nel gioco durante la fase di loading indipendentemente dall'implmentazione della classe astratta Weapon.
    - Soluzione: ho deciso di usare il metodo Factory che permette di separare il codice della costruzione dell'oggetto dal codice che lo adrà ad utilizzare.
    - Schema:

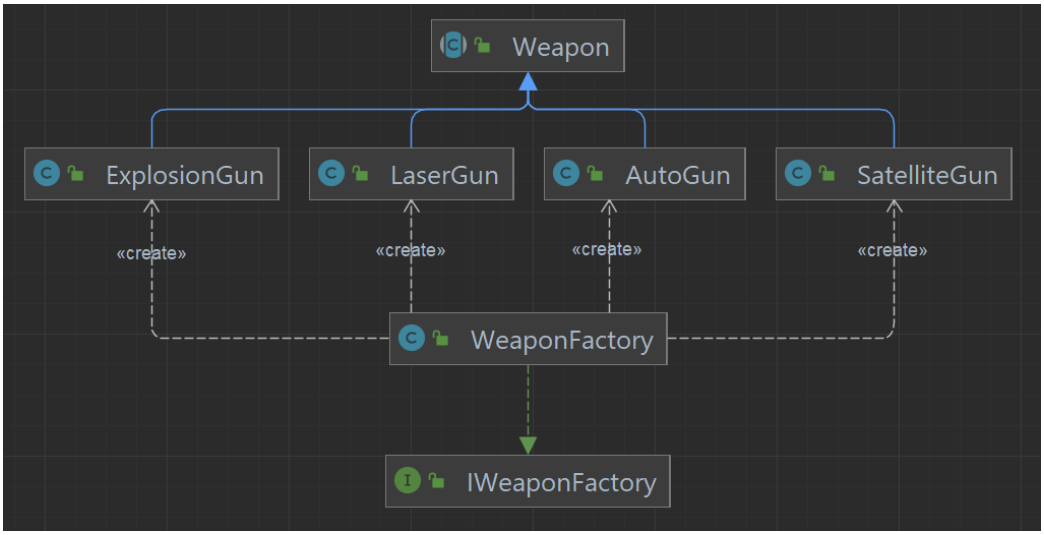


Figura 2.2: Schema UML dell'implementazione del pattern *Factory*

2. Strategy Pattern: CollisionBehavior

    - Problema: gestire le collisioni dei GameObject presenti in gioco in modo efficiente

- Soluzione ho deciso di usare lo Strategy Pattern che permette di definire una serie di algoritmi in classi separate e di poterli inserire in modo intercambiabile all'interno di altre classi. In questo modo per assegnare lo stesso comportamento a più di una classe non sarà necessario scrivere lo stesso codice più volte o usare l'ereditarietà. Inoltre gli algoritmi alla base di ciascun comportamento sono facilmente mantenibili e modificabili.
  Nella versiona attuale di gioco ho creato 3 algoritmi ciascuno gestito da una classe che implementa l'interfaccia ICollisionBehavior. Un esempio dell'utilizzo di queste classi lo si può trovare nella gestione di tutte le Bullet. Di default tutte le classi che estendono la classe astratta Bullet avranno l'algoritmo NoCollisionBehavior, mentre in AutoBullet verrà usato l'algoritmo presente in RemoveOnCollisionBehavior.
- Schema:

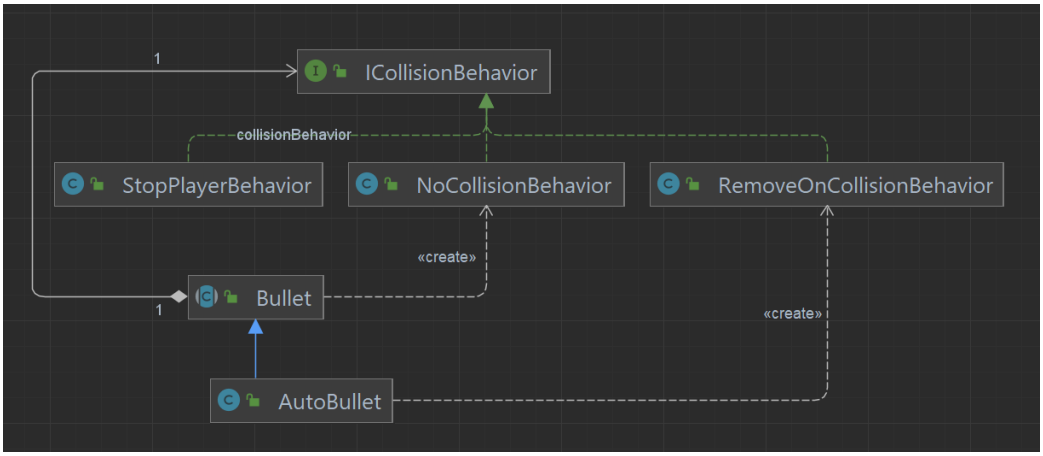


Figura 2.3: Schema UML dell'implementazione del pattern *Strategy*.

3. Template Method Pattern

- Problema: implmentare l'algoritmo che permetta a ciascuna arma di sparare
- Soluzione: ho deciso di utilizzare il Template Method in quanto il comportamento di ciascuna arma è unico e non verrà usato altrove. La classe astratta Weapon contiene l'algoritmo che permette alle armi che estenderanno di sparare in modo ciclico, ma il metodo che richiama lo sparo stesso è astratto e richiede di essere implmentato.
- Schema:

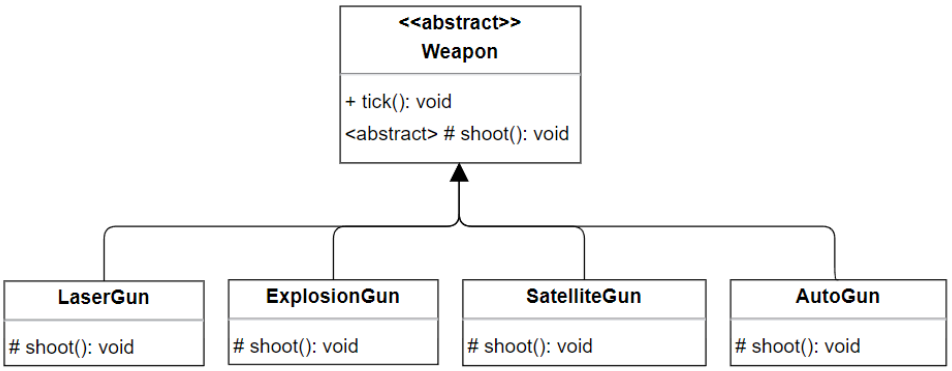


Figura 2.4: Schema UML dell'implementazione del pattern *Template*

Pierangelo Motta:

1. Problematica: configurare metodi comuni a diversi tipi di mostri.

    - E' stata utilizzata la abstract class "Monster" (che implementa l'interfaccia IMonster) per definire i metodi comuni a tutte le classi che estendono la abstract class: "Triangle", "Rect", "Rhombus" e "Ball"(aggiunta in seguito).

      Nel caso di comportamenti diversi da parte di alcuni mostri rispetto ai metodi definiti nella classe astratta, si è utilizzato l'override del metodo. Ad esempio il percorso per raggiungere il player effettuato dal mostro di tipo "Ball" è disturbato da piccole variazioni.

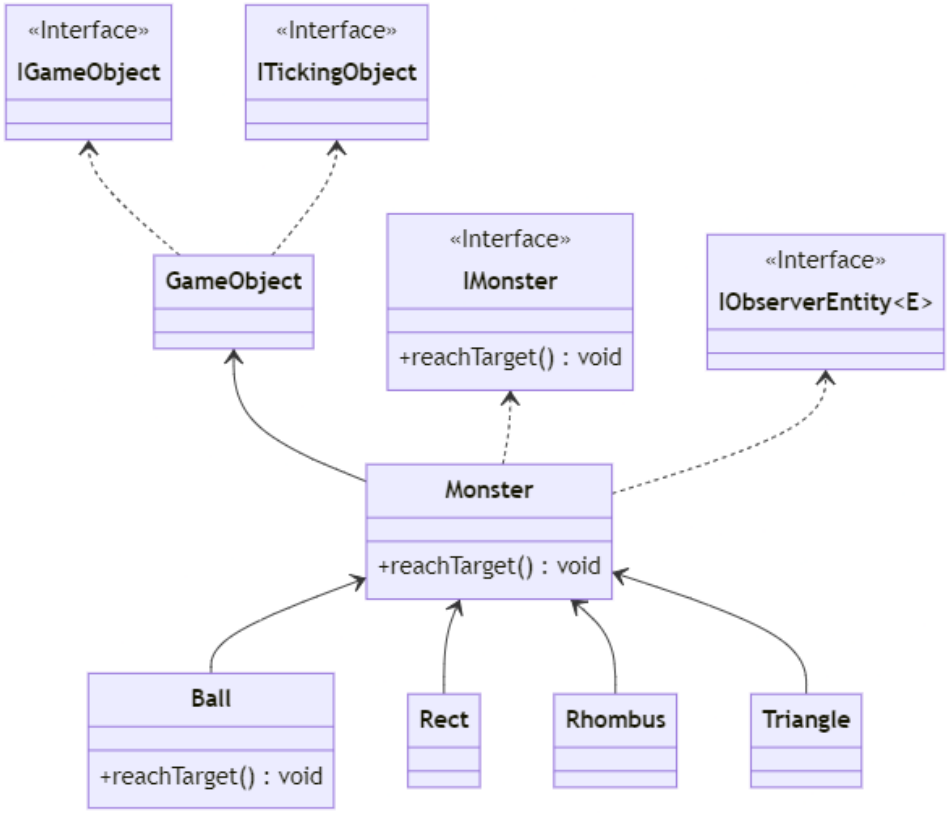


Figura 2.5: Esempio del metodo reachTarget()

2. Problematica: creare tipi di mostri diversi durante tutto l'arco temporale di gioco.

   Ho utilizzato il pattern *Factory Method* per avere delle factory che permettessero la creazione trasparente di mostri diversi. La classe MonsterSpawner utilizza le factory per generare mostri diversi e con caratteristiche diverse in base allo scorrere del tempo.

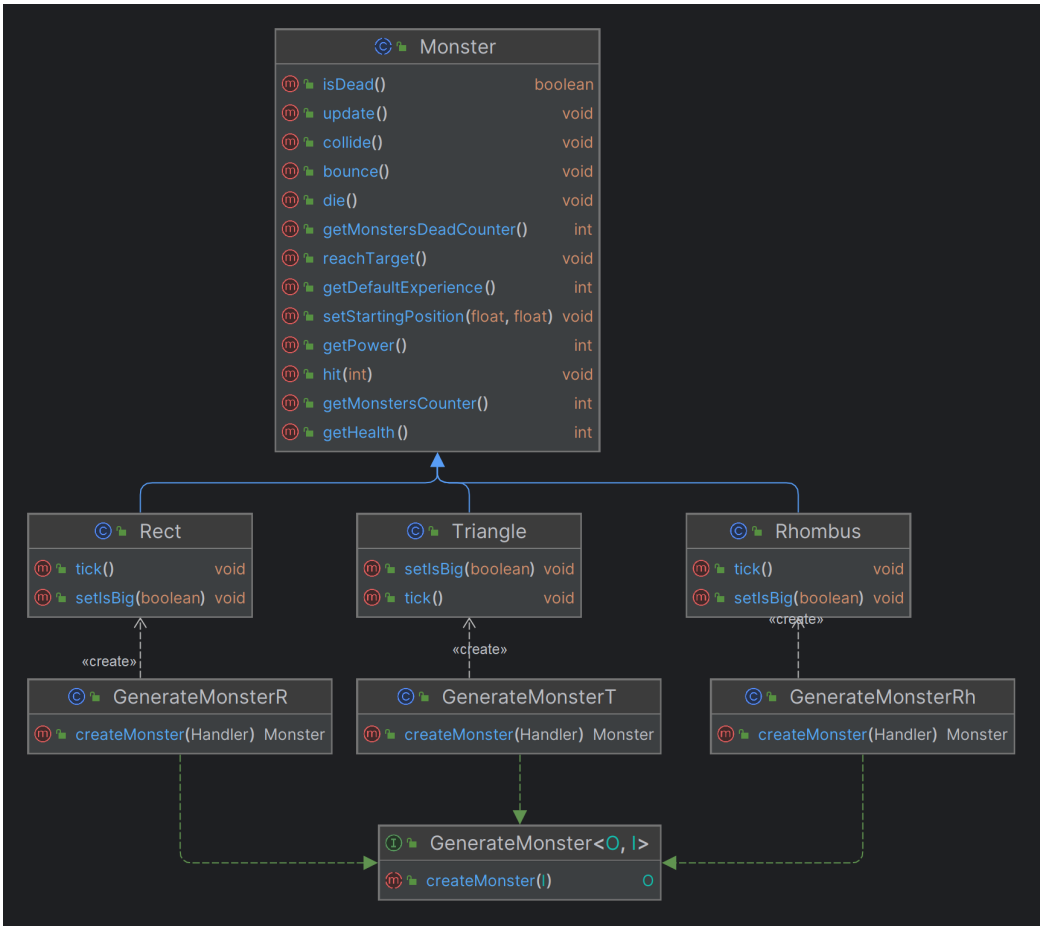


Figura 2.6: Schema UML dell'implementazione, con rappresentate le entità principali ed i rapporti fra loro

3. Problematica: posizionare correttamente i mostri alla "nascita" e fare in modo che raggiungano il player. Per la soluzione occorrono due condizioni legate dalla posizione del player nella mappa di gioco:

   - i mostri devono conoscere la posizione del player in ogni istante utile in quanto devono cercare di raggiungerlo per attaccarlo;
   - i mostri devono nascere in una posizione casuale in un'area delimitata da due cerchi concentrici (di distanza minima e massima definibili) con al centro il player.

   Nel primo caso ho implementato il pattern *Observer*, in maniera tale che ogni mostro creato si registri in una lista di *observers* del player(che

è *Observable*). Ad ogni tick() temporale il player *notifica* tutti gli osservatori.

Ho reso generica l'interfaccia IObserverEntity <E> in maniera tale che anche altri oggetti oltre al player possano essere osservati. I metodi dell'interfaccia IObservable hanno come parametro una *"bounded wildcard"* che rappresenta la possibilità di utilizzare un oggetto che appartenga ad una sottoclasse di GameObject.

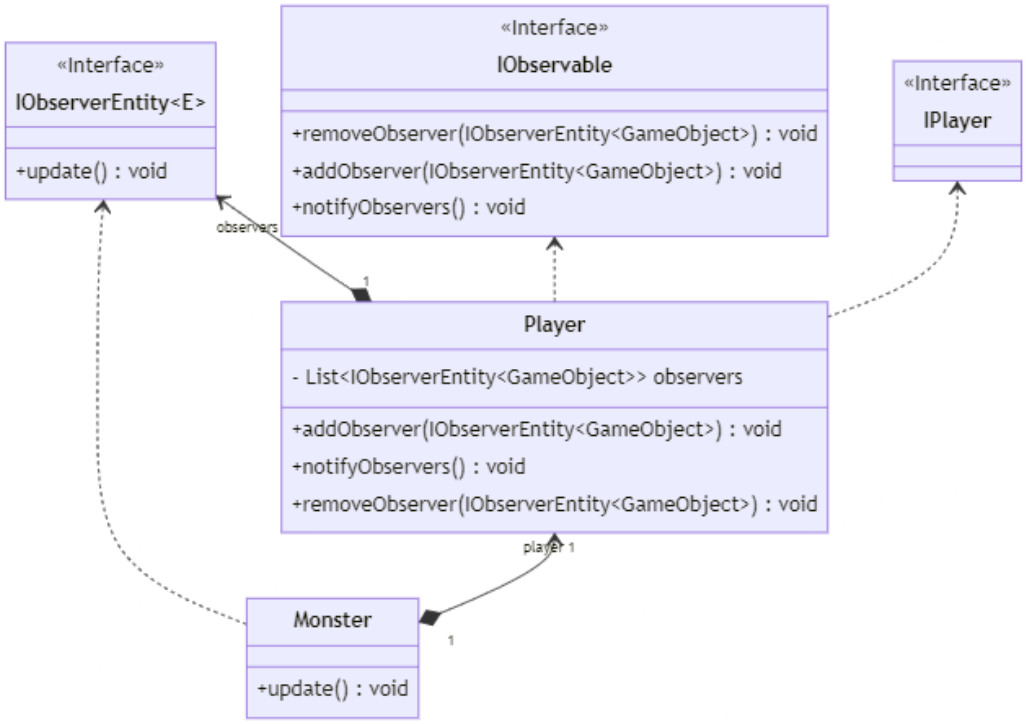


Figura 2.7: Schema UML dell'implementazione del pattern *Observer*

Nel secondo caso si è utilizzata una funzione di utilità Func.randomPoint() predisposta da Sergio per il calcolo delle coordinate nell'area compresa fra i due cerchi.

4. Permettere al player di raccogliere "Experience" anche a distanza, attraverso quello che sembra una sorta di magnetismo.

- Estendendo la funzionalità del punto precedente è stato possibile implementare in maniera agevole la funzionalità alla classe "Experience", in un secondo tempo, attraverso l'interfaccia IObserverEntity:

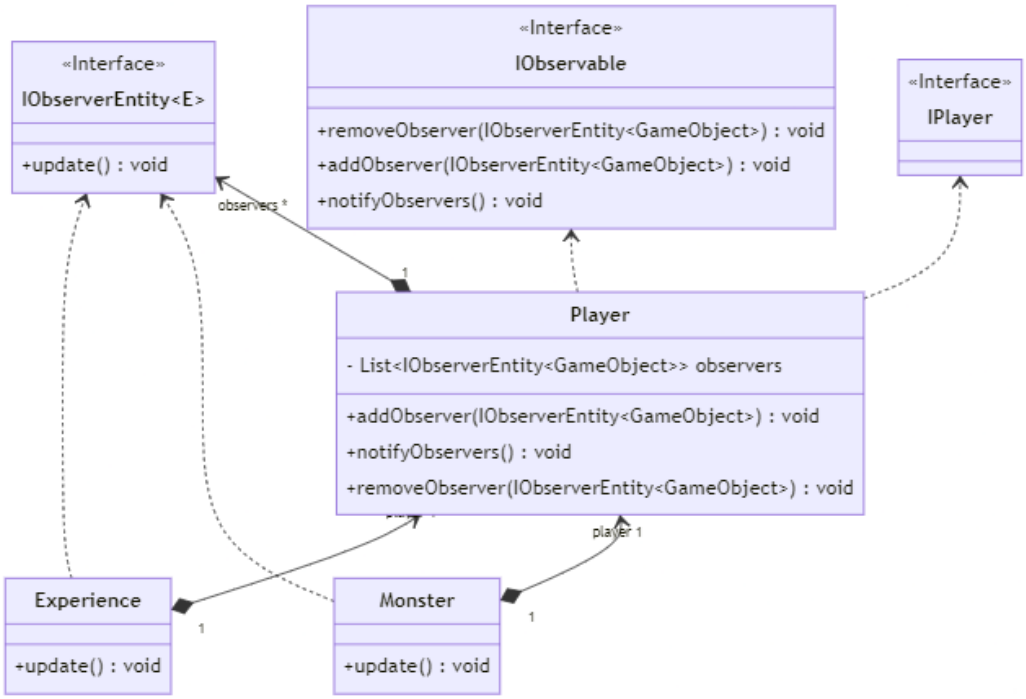


Figura 2.8: Schema UML dell'implementazione del pattern *Observer* per "Monster" ed "Experience"

- Il metodo reachTarget() della classe Experience è stato sovrascritto perchè il player inizia a raccogliere "pillole di esperienza" sormontandole. Con l'aumentare del livello raggiunto aumenta proporzionalmente la distanza da cui il player può raccogliere "pillole" con il solo avvicinarsi.

5. Problema: creare un'ondata finale di mostri per concludere il gioco.

   - ho creato una funzione flood() che, come le altre funzioni che determinano la creazione di mostri, viene eseguita un'unica volta

(nella classe MonsterSpawner) in base al tempo trascorso da inizio gioco. Al suo interno, viene utilizzato uno *stream* continuo che genera un determinato tipo di mostri, fino alla fine del gioco.

6. Drop di oggetti diversi alla morte di mostri
    - Alla morte di ogni mostro può essere rilascita (drop) una "pillola di esperienza", una "pillola di vita" o un mostro di tipo "Ball".
    - La classe Drop, che implementa l'interfaccia IDrop, contiene la strategia con cui viene selezionata la tipologia di oggetto di cui fare il drop.
    - Schema
    - Pattern
7. Problema: foreach crash??
    - Soluzione CopyOnWriteArrayList
    - Schema
    - Pattern

Thomas Testa:

1. Problema 1
    - Soluzione
    - Schema
    - Pattern
2. Problema 2
    - Soluzione
    - Schema
    - Pattern

In questa sezione si possono approfondire alcuni elementi del design con maggior dettaglio. Mentre ci attendiamo principalmente (o solo) interfacce negli schemi UML delle sezioni precedenti, in questa sezione è necessario scendere in maggior dettaglio presentando la struttura di alcune sottoparti rilevanti dell'applicazione. È molto importante che, descrivendo la soluzione ad un problema, quando possibile si mostri che non si è re-inventata la ruota

ma si è applicato un design pattern noto. Che si sia utilizzato (o riconosciuto) o meno un pattern noto, è comunque bene definire qual è il problema che si è affrontato, qual è la soluzione messa in campo, e quali motivazioni l'hanno spinta. È assolutamente inutile, ed è anzi controproducente, descrivere classe-per-classe (o peggio ancora metodo-per-metodo) com'è fatto il vostro software: è un livello di dettaglio proprio della documentazione dell'API (deducibile dalla Javadoc).

È necessario che ciascun membro del gruppo abbia una propria sezione di design dettagliato, di cui sarà il solo responsabile. Ciascun autore dovrà spiegare in modo corretto e giustamente approfondito (non troppo in dettaglio, non superficialmente) il proprio contributo. È importante focalizzarsi sulle scelte che hanno un impatto positivo sul riuso, sull'estensibilità, e sulla chiarezza dell'applicazione. Esattamente come nessun ingegnere meccanico presenta un solo foglio con l'intero progetto di una vettura di Formula 1, ma molteplici fogli di progetto che mostrano a livelli di dettaglio differenti le varie parti della vettura e le modalità di connessione fra le parti, così ci aspettiamo che voi, futuri ingegneri informatici, ci presentiate prima una visione globale del progetto, e via via siate in grado di dettagliare le singole parti, scartando i componenti che non interessano quella in esame. Per continuare il parallelo con la vettura di Formula 1, se nei fogli di progetto che mostrano il design delle sospensioni anteriori appaiono pezzi che appartengono al volante o al turbo, c'è una chiara indicazione di qualche problema di design.

Si divida la sezione in sottosezioni, e per ogni aspetto di design che si vuole approfondire, si presenti:

1. : una breve descrizione in linguaggio naturale del problema che si vuole risolvere, se necessario ci si può aiutare con schemi o immagini;

2. : una descrizione della soluzione proposta, analizzando eventuali alternative che sono state prese in considerazione, e che descriva pro e contro della scelta fatta;

3. : uno schema UML che aiuti a comprendere la soluzione sopra descritta;

4. : se la soluzione è stata realizzata utilizzando uno o più pattern noti, si spieghi come questi sono reificati nel progetto (ad esempio: nel caso di Template Method, qual è il metodo template; nel caso di Strategy, quale interfaccia del progetto rappresenta la strategia, e quali sono le sue implementazioni; nel caso di Decorator, qual è la classe astratta che fa da Decorator e quali sono le sue implementazioni concrete; eccetera);

La presenza di pattern di progettazione *correttamente utilizzati* è valutata molto positivamente. L'uso inappropriato è invece valutato negativamente: a tal proposito, si raccomanda di porre particolare attenzione all'abuso di Singleton, che, se usato in modo inappropriato, è di fatto un anti-pattern.

Elementi positivi

- Ogni membro del gruppo discute le proprie decisioni di progettazione, ed in particolare le azioni volte ad anticipare possibili cambiamenti futuri (ad esempio l'aggiunta di una nuova funzionalità, o il miglioramento di una esistente).
- Si mostrano le principali interazioni fra le varie componenti che collaborano alla soluzione di un determinato problema.
- Si identificano, utilizzano *appropriatamente*, e descrivono diversi design pattern.
- Ogni membro del gruppo identifica i pattern utilizzati nella sua sottoparte.
- Si mostrano gli aspetti di design più rilevanti dell'applicazione, mettendo in luce la maniera in cui si è costruita la soluzione ai problemi descritti nell'analisi.
- Si tralasciano aspetti strettamente implementativi e quelli non rilevanti, non mostrandoli negli schemi UML (ad esempio, campi privati) e non descrivendoli.
- Ciascun elemento di design identificato presenta una piccola descrizione del problema calato nell'applicazione, uno schema UML che ne mostra la concretizzazione nelle classi del progetto, ed una breve descrizione della motivazione per cui tale soluzione è stata scelta, specialmente se è stato utilizzato un pattern noto. Ad esempio, se si dichiara di aver usato Observer, è necessario specificare chi sia l'observable e chi l'observer; se si usa Template Method, è necessario indicare quale sia il metodo template; se si usa Strategy, è necessario identificare l'interfaccia che rappresenta la strategia; e via dicendo.

Elementi negativi

- Il design del modello risulta scorrelato dal problema descritto in analisi.

- Si tratta in modo prolisso, classe per classe, il software realizzato, o comunque si riduce la sezione ad un mero elenco di quanto fatto.
- Non si presentano schemi UML esemplificativi.
- Non si individuano design pattern, o si individuano in modo errato (si spaccia per design pattern qualcosa che non lo è).
- Si utilizzano design pattern in modo inopportuno. Un esempio classico è l'abuso di Singleton per entità che possono essere univoche ma non devono necessariamente esserlo. Si rammenta che Singleton ha senso nel secondo caso (ad esempio `System` e `Runtime` sono singleton), mentre rischia di essere un problema nel secondo. Ad esempio, se si rendesse singleton il motore di un videogioco, sarebbe impossibile riusarlo per costruire un server per partite online (dove, presumibilmente, si gestiscono parallelamente più partite).
- Si producono schemi UML caotici e difficili da leggere, che comprendono inutili elementi di dettaglio.
- Si presentano schemi UML con classi (nel senso UML del termine) che "galleggiano" nello schema, non connesse, ossia senza relazioni con il resto degli elementi inseriti.
- Si tratta in modo inutilmente prolisso la divisione in package, elencando ad esempio le classi una per una.

Esempio minimale (e quindi parziale) di sezione di progetto con UML ben realizzati

Personalità intercambiabili

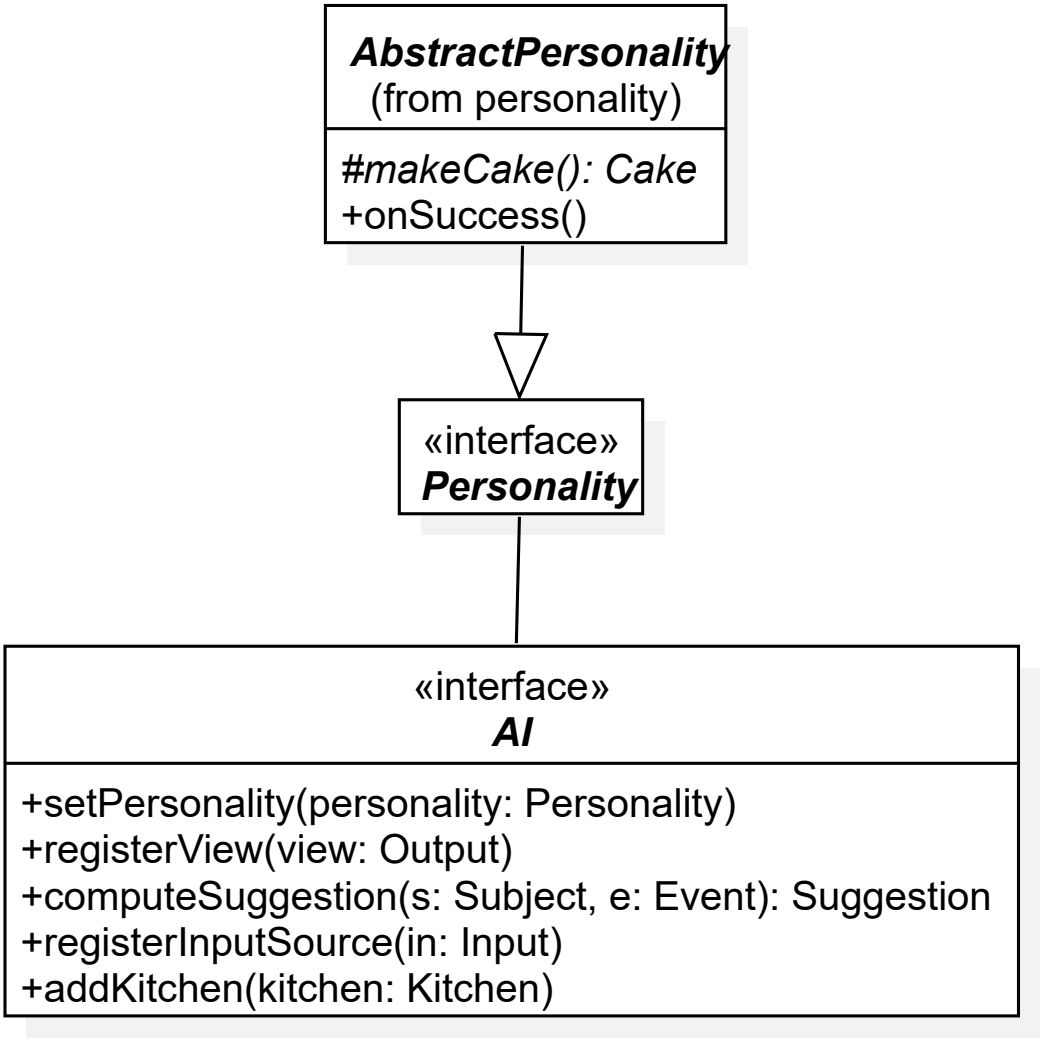


Figura 2.9: Rappresentazione UML del pattern Strategy per la personalità di GLaDOS

Problema GLaDOS ha più personalità intercambiabili, la cui presenza deve essere trasparente al client.

Soluzione Il sistema per la gestione della personalità utilizza il *pattern Strategy*, come da Figura 2.9: le implementazioni di `Personality` possono

essere modificate, e la modifica impatta direttamente sul comportamento di GLaDOS.

Riuso del codice delle personalità

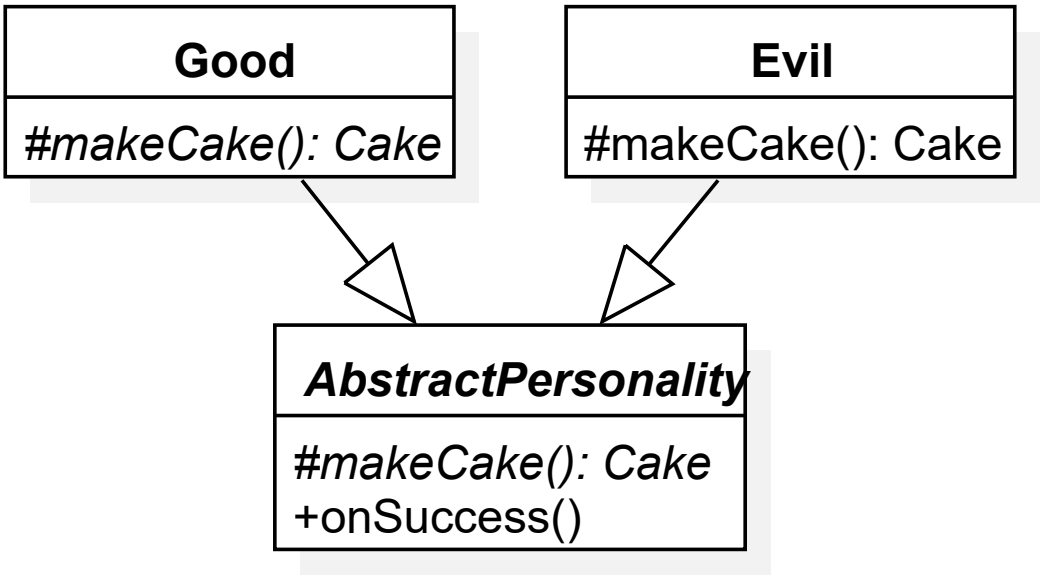


Figura 2.10: Rappresentazione UML dell'applicazione del pattern Template Method alla gerarchia delle Personalità

Problema In fase di sviluppo, sono state sviluppate due personalità, una buona ed una cattiva. Quella buona restituisce sempre una torta vera, mentre quella cattiva restituisce sempre la promessa di una torta che verrà in realtà disattesa. Ci si è accorti che diverse personalità condividevano molto del comportamento, portando a classi molto simili e a duplicazione.

Soluzione Dato che le due personalità differiscono solo per il comportamento da effettuarsi in caso di percorso completato con successo, è stato utilizzato il *pattern template method* per massimizzare il riuso, come da Figura 2.10. Il metodo template è `onSuccess()`, che chiama un metodo astratto e protetto `makeCake()`.

Gestione di output multipli

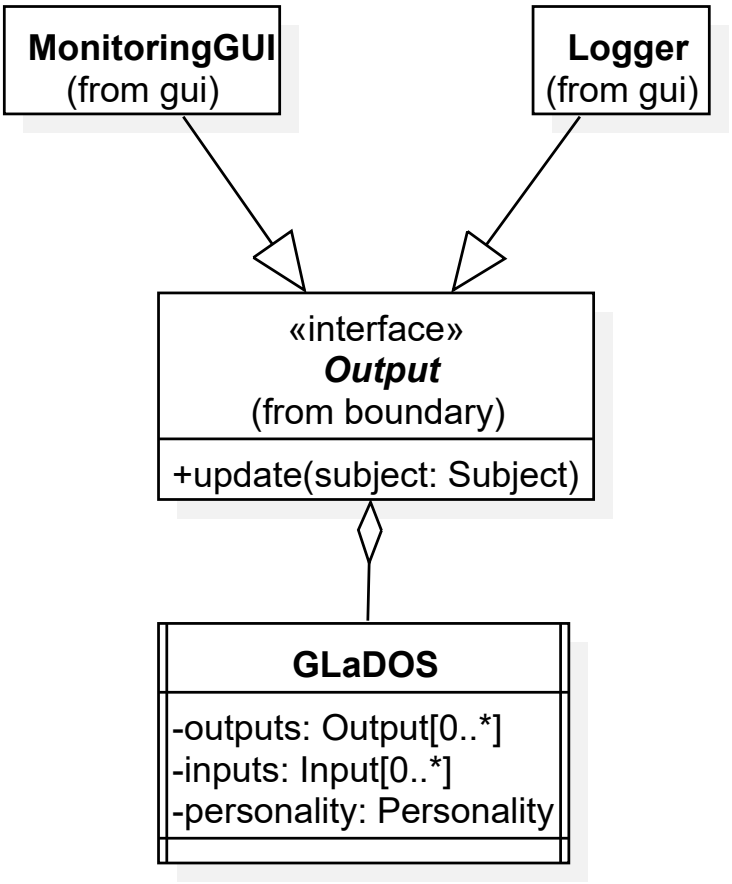


Figura 2.11: Il pattern Observer è usato per consentire a GLaDOS di informare tutti i sistemi di output in ascolto

Problema Il sistema deve supportare output multipli. In particolare, si richiede che vi sia un logger che stampa a terminale o su file, e un'interfaccia grafica che mostri una rappresentazione grafica del sistema.

Soluzione Dato che i due sistemi di reporting utilizzano le medesime informazioni, si è deciso di raggrupparli dietro l'interfaccia `Output`. A questo punto, le due possibilità erano quelle di far sì che `GLaDOS` potesse pilotarle entrambe. Invece di fare un sistema in cui questi output sono obbligatori e connessi, si è deciso di usare maggior flessibilità (anche in vista di future estensioni) e di adottare una comunicazione uno-a-molti fra `GLaDOS` ed i sistemi di output. La scelta è quindi ricaduta sul *pattern Observer*: `GLaDOS` è

observable, e le istanze di `Output` sono observer. Il suo utilizzo è esemplificato in Figura 2.11

Contro-esempio: pessimo diagramma UML

In Figura 2.12 è mostrato il modo **sbagliato** di fare le cose. Questo schema è fatto male perché:

- È caotico.
- È difficile da leggere e capire.
- Vi sono troppe classi, e non si capisce bene quali siano i rapporti che intercorrono fra loro.
- Si mostrano elementi implementativi irrilevanti, come i campi e i metodi privati nella classe `AbstractEnvironment`.
- Se l'intenzione era quella di costruire un diagramma architetturale, allora lo schema è ancora più sbagliato, perché mostra pezzi di implementazione.
- Una delle classi, in alto al centro, galleggia nello schema, non connessa a nessuna altra classe, e di fatto costituisce da sola un secondo schema UML scorrelato al resto
- Le interfacce presentano tutti i metodi e non una selezione che aiuti il lettore a capire quale parte del sistema si vuol mostrare.

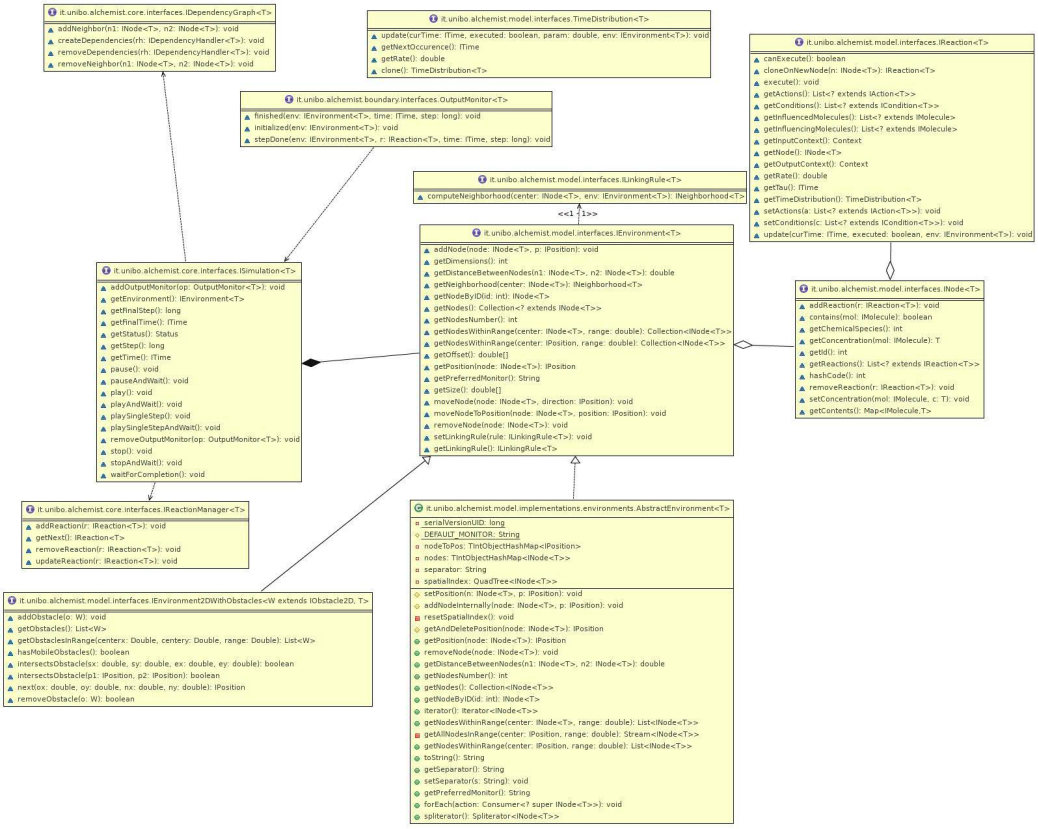


Figura 2.12: Schema UML mal fatto e con una pessima descrizione, che non aiuta a capire. Don't try this at home.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Motta:

- settaggio ambiente di test Junit
- TestMonsterCreation: per verificare la creazione di tipi diversi di entità utilizzando le factory
- TestMonsterCreation: per verificare l'esistenza dei mostri attraverso il corretto conteggio degli stessi
- TestMonsterDie: verifica in step successivi della presenza nel gioco del "Player" e di un numero N di mostri. Alla morte di tutti i mostri verifica della presenza delle altre possibili tipologie di entità nate di conseguenza: "Experience", "Life" oppure "Ball" (mostro zombie).

Sergio Dobrianskiy:

- TestBullets: per verificare la creazione e il corretto funzionamento delle 4 classi che estendono Bullet.
- TestWeapons: per verificare la creazione e il corretto funzionamento delle 4 classi che estendono Weapon.
- TestCollisions: per verificare la creazione e il corretto funzionamento della meccanica di collisione tra vari GameObject. Assicura anche che elementi come AutoBullet vengano rimossi dopo che è accaduta una collisione, mentre elementi come Satellite rimangano in gioco.

Testa:

- xxxxxxxxxxxx
- yyyyyyyyyyyyyyy

Per quanto riguarda questo progetto è considerato sufficiente un test minimale, a patto che sia completamente automatico. Test che richiedono l'intervento da parte dell'utente sono considerati *negativamente* nel computo del punteggio finale.

Elementi positivi

- Si descrivono molto brevemente i componenti che si è deciso di sottoporre a test automatizzato.
- Si utilizzano suite specifiche (e.g. JUnit) per il testing automatico.

Elementi negativi

- Non si realizza alcun test automatico.
- La non presenza di testing viene aggravata dall'adduzione di motivazioni non valide. Ad esempio, si scrive che l'interfaccia grafica non è testata automaticamente perché è *impossibile* farlo[1].
- Si descrive un testing di tipo manuale in maniera prolissa.
- Si descrivono test effettuati manualmente che sarebbero potuti essere automatizzati, ad esempio scrivendo che si è usata l'applicazione manualmente.
- Si descrivono test non presenti nei sorgenti del progetto.
- I test, quando eseguiti, falliscono.

[1]Testare in modo automatico le interfacce grafiche è possibile (si veda, come esempio, `https://github.com/TestFX/TestFX`), semplicemente nel corso non c'è modo e tempo di introdurvi questo livello di complessità. Il fatto che non vi sia stato insegnato come farlo non implica che sia impossibile!

3.2 Metodologia di lavoro

Il gruppo ha lavorato in modo coeso sia durante la scelta del tipo progetto da realizzare che successivamente per la stesura di una lista di funzionalità minime da portare a termine. Ciascun membro ha scelto da questa lista le parti sulle quali avrebbe lavorato. Fatto ciò, sempre in gruppo, abbiamo deciso un design architetturale da seguire.

Lungo il corso di tutto il progetto sono stati utilizzati un gruppo Whatsapp per la comunicazione quotidiana, un canale discord per le riunioni settimanali e un documento condiviso su Google Drive per appunti e materiale utile. Dato che i membri del gruppo vivono lontani tra di loro almeno due sono lavoratori full time non è stato possibile organizzare incontri dal vivo.

Lavorando al progetto ci siamo resi conto che il modello architetturale previsto nelle prime fasi era inadeguato. Questo problema è stato risolto agilmente grazie alla comunicazione frequente tra i membri.

Un'ulteriore difficotà è nata a causa di problemi personali del collega Matteo Trezza che, nonostante i tentativi di aiuto, ha faticato a stare al passo con il resto del gruppo e alla fine ha dovuto ritirarsi. Dato il bisogno di portare avanti le parti di Matteo, in sua assenza abbiamo provveduto a spartircele in base alle necessità e preferenze di ciascuno dei membri rimanenti.

Per condividere il codice è stato usato fin da subito Git. Ci siamo impegnati a non lavorare direttamente sul branch *master* e di creare per ciascuna feature un branch separato. Per consentire a tutti di avere un codice aggiornato abbiamo cercato di fare dei merge frequenti.

Di seguto è indicata nel dettaglio la suddivisione dei compiti:

Sergio Dobrianskiy:

- Gestione delle armi e dei proiettili
- Gestione delle collisioni (java.awt)
- Architettura dei GameObject e TickingObject
- Gestione caricamento e loop di gioco (Loader e Handler)
- Gestione delle texture/sprite

- Gestione dei controlli del giocatore

  Inizialmente avevo scelto di occuparmi solo delle armi, dei proiettili e delle collisioni, ma data la necessità e l'interesse personale ho finito per occuparmi in modo massiccio anche degli altri punti indicati nella lista.

  Per la gestione delle collisioni ho utilizzato le funzioni geometriche presenti in *java.awt*.

  Come indicato nella sezione apposita, ho adattaro del codice preso dal progetto OOP21-ciccio-pier per la creazione della classe Texture e la gestione degli sprite.

Ci aspettiamo, leggendo questa sezione, di trovare conferma alla divisione operata nella sezione del design di dettaglio, e di capire come è stato svolto il lavoro di integrazione. **Andrà realizzata una sotto-sezione separata per ciascuno studente** che identifichi le porzioni di progetto sviluppate, separando quelle svolte in autonomia da quelle sviluppate in collaborazione. Diversamente dalla sezione di design, in questa è consentito elencare package/classi, se lo studente ritiene sia il modo più efficace di convogliare l'informazione. Si ricorda che l'impegno deve giustificare circa 40-50 ore di sviluppo (è normale e fisiologico che approssimativamente la metà del tempo sia impiegata in analisi e progettazione).

Elementi positivi

- Si identifica con precisione il ruolo di ciascuno all'interno del gruppo, ossia su quale parte del progetto ciascuno dei componenti si è concentrato maggiormente.
- La divisione dei compiti è equa, ossia non vi sono membri del gruppo che hanno svolto molto più lavoro di altri.
- La divisione dei compiti è coerente con quanto descritto nelle parti precedenti della relazione.

- La divisione dei compiti è realistica, ossia le dipendenze fra le parti sviluppate sono minime.
- Si identifica quale parte del software è stato sviluppato da tutti i componenti insieme.
- Si spiega in che modo si sono integrate le parti di codice sviluppate separatamente, evidenziando eventuali problemi. Ad esempio, una strategia è convenire sulle interfacce da usare (ossia, occuparsi insieme di stabilire l'architettura) e quindi procedere indipendentemente allo sviluppo di parti differenti. Una possibile problematica potrebbe essere una dimenticanza in fase di design architetturale che ha costretto ad un cambio e a modifiche in fase di integrazione. Una situazione simile è la norma nell'ingegneria di un sistema software non banale, ed il processo di progettazione top-down con raffinamento successivo è il così detto processo "a spirale".
- Si descrive in che modo è stato impiegato il DVCS.

Elementi negativi

- Non si chiarisce chi ha fatto cosa.
- C'è discrepanza fra questa sezione e le sezioni che descrivono il design dettagliato.
- Tutto il progetto è stato svolto lavorando insieme invece che assegnando una parte a ciascuno.
- Non viene descritta la metodologia di integrazione delle parti sviluppate indipendentemente.
- Uso superficiale del DVCS.

3.3 Note di sviluppo

Questa sezione, come quella riguardante il design dettagliato va svolta **singolarmente da ogni membro del gruppo**. Nella prima parte, ciascuno dovrà mostrare degli esempi di codice particolarmente ben realizzati, che dimostrino proefficienza con funzionalità avanzate del linguaggio e capacità di spingersi oltre le librerie mostrate a lezione.

- **Elencare** (fare un semplice elenco per punti, non un testo!) le feature *avanzate* del linguaggio e dell'ecosistema Java che sono state utilizzate. Le feature di interesse sono:
  - Progettazione con generici, ad esempio costruzione di nuovi tipi generici, e uso di generici bounded. L'uso di classi generiche di libreria non è considerato avanzato.
  - Uso di lambda expressions
  - Uso di `Stream`, di `Optional` o di altri costrutti funzionali
  - Uso di reflection
  - Definizione ed uso di nuove annotazioni
  - Uso del Java Platform Module System
  - Uso di parti della libreria JDK non spiegate a lezione (networking, compressione, parsing XML, eccetera...)
  - Uso di librerie di terze parti (incluso JavaFX): Google Guava, Apache Commons...
- Si faccia molta attenzione a non scrivere banalità, elencando qui features di tipo "core", come le eccezioni, le enumerazioni, o le inner class: nessuna di queste è considerata avanzata.
- Per ogni feature avanzata, mostrata, includere:
  - Nome della feature
  - Permalink GitHub al punto nel codice in cui è stata utilizzata

Per facilitarci nello sviluppo del gioco, soprattutto nelle fasi iniziali, abbiamo seguito dei tutorial oppure sono state usate delle funzionalità di un altro progetto, di seguito viene fortina una lista:

- Per la creazione da zero del motore di gioco e del motore grafico abbiamo preso spunto da due tutorial presi da Youtube:
  - Java Game Programming Wizard Course
  - Java Programming Let's Build a Zombie Game

  In particolare con l'aiuto di questi due tutorial sono state create le classi Game, GameObject, Handler, KeyInput usate per ottenere una versione rudimentale del motore di gioco e del suo motore grafico. Da quel momento in poi le classi sono state fortemente riviste nelle funzionalità e riadattate ad una logica OOP.

- Per il caricamento e la gestione delle texture/sprite che avviene in particolare nelle classi Texture e Loader e la creazione di Block durante il cariamento della mappa è stato preso e riadattato del codice dal progetto OOP21-ciccio-pier consegnato da un altro gruppo per l'esame di Programmazione ad Oggetti.

In questa sezione, *dopo l'elenco*, vanno menzionati ed attributi con precisione eventuali pezzi di codice "riadattati" (o scopiazzati...) da Internet o da altri progetti, pratica che tolleriamo ma che non raccomandiamo. Si rammenta agli studenti che non è consentito partire da progetti esistenti e procedere per modifiche successive. Si ricorda anche che i docenti hanno in mano strumenti antiplagio piuttosto raffinati e che "capiscono" il codice e la storia delle modifiche del progetto, per cui tecniche banali come cambiare nomi (di classi, metodi, campi, parametri, o variabili locali), aggiungere o togliere commenti, oppure riordinare i membri di una classe vengono individuate senza problemi. Le regole del progetto spiegano in dettaglio l'approccio dei docenti verso atti gravi come il plagiarismo.

I pattern di design **non** vanno messi qui. L'uso di pattern di design (come suggerisce il nome) è un aspetto avanzato di design, non di implementazione, e non va in questa sezione.

Elementi positivi

- Si elencano gli aspetti avanzati di linguaggio che sono stati impiegati
- Si elencano le librerie che sono state utilizzate
- Per ciascun elemento, si fornisce un permalink
- Ogni permalink fa riferimento ad uno snippet di codice scritto dall'autore della sezione (i docenti verificheranno usando `git blame`)
- Se si è utilizzato un particolare algoritmo, se ne cita la fonte originale. Ad esempio, se si è usato Mersenne Twister per la generazione di numeri pseudo-random, si cita [**?**].
- Si identificano parti di codice prese da altri progetti, dal web, o comunque scritte in forma originale da altre persone. In tal senso, si ricorda che agli ingegneri non è richiesto di re-inventare la ruota continuamente: se si cita debitamente la sorgente è tollerato fare uso di di snippet di codice open source per risolvere velocemente problemi non banali. Nel caso in cui si usino snippet di codice di qualità discutibile, oltre a menzionarne l'autore originale si invitano gli studenti ad adeguare tali parti di codice agli standard e allo stile del progetto. Contestualmente, si fa presente che è largamente meglio fare uso di una libreria che copiarsi pezzi di codice: qualora vi sia scelta (e tipicamente c'è), si preferisca la prima via.

Elementi negativi

- Si elencano feature core del linguaggio invece di quelle segnalate. Esempi di feature core da non menzionare sono:
  - eccezioni;
  - classi innestate;
  - enumerazioni;
  - interfacce.
- Si elencano applicazioni di terze parti (peggio se per usarle occorre licenza, e lo studente ne è sprovvisto) che non c'entrano nulla con lo sviluppo, ad esempio:
  - Editor di grafica vettoriale come Inkscape o Adobe Illustrator;
  - Editor di grafica scalare come GIMP o Adobe Photoshop;

  - Editor di audio come Audacity;
  - Strumenti di design dell'interfaccia grafica come SceneBuilder: il codice è in ogni caso inteso come sviluppato da voi.

- Si descrivono aspetti di scarsa rilevanza, o si scende in dettagli inutili.

- Sono presenti parti di codice sviluppate originalmente da altri che non vengono debitamente segnalate. In tal senso, si ricorda agli studenti che i docenti hanno accesso a tutti i progetti degli anni passati, a Stack Overflow, ai principali blog di sviluppatori ed esperti Java, ai blog dedicati allo sviluppo di soluzioni e applicazioni (inclusi blog dedicati ad Android e allo sviluppo di videogame), nonché ai vari GitHub, GitLab, e Bitbucket. Conseguentemente, è *molto* conveniente *citare* una fonte ed usarla invece di tentare di spacciare per proprio il lavoro di altri.

- Si elencano design pattern

3.3.1 Note di sviluppo

Sergio Dobrianskiy

Funzionalità avanzate utilizzate:

- Utilizzo di Lambda e Stream
  - Permalink: `https://github.com/Sergio-Dobrianskiy/OOP22-geo-surv/blob/1d19887356d83d29b39fe301273ba64ba44ce662/src/main/java/it/unibo/geosurv/model/collisions/Collisions.java#LL33C9-L51C12`
  - Permalink: `https://github.com/Sergio-Dobrianskiy/OOP22-geo-surv/blob/5962bab31e3fd7033c2b8bd0e84b13aa7f7c85ef/src/main/java/it/unibo/geosurv/model/Handler.java#LL35C6-L35C6`

Utilizzo della libreria SLF4J

Utilizzata in vari punti. Un esempio è `https://github.com/AlchemistSimulator/Alchemist/blob/5c17f8b76920c78d955d478864ac1f11508ed9ad/alchemist-swingui/src/main/java/it/unibo/alchemist/boundary/swingui/effect/impl/EffectBuilder.java#L49`

Utilizzo di `LoadingCache` dalla libreria Google Guava

Permalink: `https://github.com/AlchemistSimulator/Alchemist/blob/d8a1799027d7d685569e15316a32e6394632ce71/alchemist-incarnation-protelis/src/main/java/it/unibo/alchemist/protelis/AlchemistExecutionContext.java#L141-L143`

Utilizzo di `Stream` e lambda expressions

Usate pervasivamente. Il seguente è un singolo esempio. Permalink: `https://github.com/AlchemistSimulator/Alchemist/blob/d8a1799027d7d685569e15316a32e6394alchemist-incarnation-protelis/src/main/java/it/unibo/alchemist/model/ProtelisIncarnation.java#L98-L120`

Scrittura di metodo generico con parametri contravarianti

Permalink: `https://github.com/AlchemistSimulator/Alchemist/blob/d8a1799027d7d685569e15316a32e6394632ce71/alchemist-incarnation-protelis/src/main/java/it/unibo/alchemist/protelis/AlchemistExecutionContext.java#L141-L143`

Protezione da corse critiche usando `Semaphore`

Permalink: `https://github.com/AlchemistSimulator/Alchemist/blob/d8a1799027d7d685569e15316a32e6394632ce71/alchemist-incarnation-protelis/src/main/java/it/unibo/alchemist/model/ProtelisIncarnation.java#L388-L440`

Capitolo 4

Commenti finali

In quest'ultimo capitolo si tirano le somme del lavoro svolto e si delineano eventuali sviluppi futuri.

Nessuna delle informazioni incluse in questo capitolo verrà utilizzata per formulare la valutazione finale, a meno che non sia assente o manchino delle sezioni obbligatorie. Al fine di evitare pregiudizi involontari, l'intero capitolo verrà letto dai docenti solo dopo aver formulato la valutazione.

4.1 Autovalutazione e lavori futuri

Sergio Dobrianskiy:

Ho cercato i membri per formare il gruppo e proposto l'idea di gioco sulla quale lavorare. Durante lo svolgimento del progetto ho cercato di indirizzare ciascun membro su obbiettivi a breve termine che mi sembravano più importanti, in particolar modo durante le fasi iniziali ho insistito sulle funzionalità necessarie a tutto gruppo per poter procedere nello sviluppo. Quando mi sono reso conto che il collega Matteo Trezza non avrebbe portato a termine la sua parte di progetto mi sono occupato di una buona parte delle sue funzionalità minime.

Sono soddisfatto del lavoro fatto da me e dal gruppo, l'organizzazione e il team work sono stati ottimi. Inoltre creare da zero un gioco, anche se semplice, mi ha appassionato e mi ha permesso di imparare molto. Il mio punto debole è stata la totale inesperienza con lo sviluppo di un gioco che mi ha reso impossibile avere un'idea chiara sulla scaletta di cosa sviluppare nel medio-lungo periodo. Inoltre la scarsa famigliarità iniziale con i pattern di programmazione mi hanno costretto a un refactoring in corso d'opera.

Se dovessi continuare a lavorare al progetto vorrei lavorare su una versione mobile e riscrivere in modo migliore la parte del motore di gioco.

È richiesta una sezione per ciascun membro del gruppo, obbligatoriamente. Ciascuno dovrà autovalutare il proprio lavoro, elencando i punti di forza e di debolezza in quanto prodotto. Si dovrà anche cercare di descrivere *in modo quanto più obiettivo possibile* il proprio ruolo all'interno del gruppo. Si ricorda, a tal proposito, che ciascuno studente è responsabile solo della propria sezione: non è un problema se ci sono opinioni contrastanti, a patto che rispecchino effettivamente l'opinione di chi le scrive. Nel caso in cui si pensasse di portare avanti il progetto, ad esempio perché effettivamente impiegato, o perché sufficientemente ben riuscito da poter esser usato come dimostrazione di esser capaci progettisti, si descriva brevemente verso che direzione portarlo.

Appendice A

Guida utente

Il player deve sfuggire ai mostri muovendosi con le le frecce direzionali o con i comuni tasti W (su) A(sinistra) S(giù) D(destra).

Il player inizia il gioco con un'arma a disposizione e, all'aumentare di livello, vengono aggiunte nuove armi e tutte si potenziano. Il livello cresce se il player raccoglie gemme blu ("pillole di esperienza") create dalla morte dei mostri. Alla morte di un mostro potrebbero anche essere rilasciati una gemma verde("Life") che se raccolta fornisce un recupero di vita al player oppure potrebbe essere creato un nuovo tipo di mostro("Ball").

Il gioco finisce se il player esauriusce la vita a disposizione (il player muore) o dopo un minuto all'arrivo di un'ondata finale di mostri (il player vince).

Come indicato nella schermata iniziale:

- p) mette il gioco in pausa;
- g) attiva la modalità di debug che mostra informazioni sugli oggetti presenti.