Relazione del progetto "coffeBreak"

Grazia Bochdanovits de Kavna
Alessandro Rebosio
Filippo Riccioti

30 giugno 2025

Indice

1 Analisi **2**
1.1 Descrizione e requisiti 2
1.2 Modello del Dominio 4

2 Design **7**
2.1 Architettura 7
2.2 Design dettagliato 9

3 Sviluppo **19**
3.1 Testing automatizzato 19
3.2 Note di sviluppo 22
3.2.1 Alessandro Rebosio 22
3.2.2 Istruzioni note di sviluppo 22

4 Commenti finali **26**
4.1 Autovalutazione e lavori futuri 26
4.2 Difficoltà incontrate e commenti per i docenti 26

A Guida utente **28**
A.1 Menu Principale 28
A.2 Durante il Gioco 28
A.3 Menu di Pausa 28
A.4 Game Over 28

B Esercitazioni di laboratorio **29**
B.1 alessandro.rebosio@studio.unibo.it 29

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software da noi sviluppato è una riproduzione del celebre videogioco arcade *Donkey Kong*, pubblicato per la prima volta nel 1981 da Nintendo.

Il protagonista è Mario, il cui obiettivo è salvare Pauline, rapita dal gigantesco gorilla Donkey Kong. Il giocatore prende il controllo di Mario e lo guida in un platform a livelli fissi, caratterizzato da strutture complesse composte da piattaforme, scale e numerosi ostacoli da superare. All'inizio di ogni partita, Mario dispone di tre vite, ogni volta che entra in contatto con un nemico, come barili rotolanti o fiamme, perde una vita e viene riposizionato all'inizio del livello corrente. La partita continua finché ci sono vite disponibili; una volta esaurite, compare la schermata di *Game Over*, che riporta l'utente al menu principale, da cui è possibile avviare una nuova partita. Il gameplay si basa su una combinazione di tempismo e precisione nei movimenti. Mario può spostarsi verso sinistra o destra, saltare per evitare i nemici e utilizzare le scale per muoversi verticalmente tra le piattaforme. L'obiettivo di ogni livello è raggiungere la cima della struttura, dove si trovano Donkey Kong e Pauline. I quattro livelli predefiniti si ripetono ciclicamente ma presentano una difficoltà crescente: con il progredire del gioco, gli ostacoli si muovono più rapidamente, i nemici diventano meno prevedibili e i percorsi sempre più complessi. Un aspetto centrale del gioco è il sistema di punteggio, che premia il giocatore per varie azioni compiute durante la partita. I punti vengono assegnati per ogni ostacolo evitato, nemico eliminato o oggetto bonus raccolto, come le monete e il martello, posizionati in punti chiave del livello. Un ruolo particolare è riservato al martello, un potenziamento temporaneo che consente a Mario di distruggere barili e nemici per alcuni secondi, offrendo un vantaggio momentaneo e un'opportunità per accumulare punti extra.

Requisiti funzionali

- **Controlli di gioco:** il giocatore deve poter controllare il personaggio principale tramite input direzionali: camminata (destra/sinistra), salto e salita/discesa scale.
- **Gestione dei livelli e ostacoli:** il sistema deve generare quattro livelli di gioco in sequenza e gestire la dinamica degli ostacoli, incluso il lancio periodico dei barili da parte di Donkey Kong.
- **Rilevamento e gestione collisioni:** il sistema deve rilevare le collisione tra il personaggio e gli ostacoli/nemici, e applicare le conseguenze previste (es. perdita di vita o bonus).
- **Sistema di punteggio e condizioni di gioco:** il sistema deve calcolare il punteggio in base a: salti sui barili, raccolta di oggetti bonue ed eliminazione di nemici con il martello. Inoltre, deve rilevare la *condizione di vittoria* quando raggiunge Paulina, e la *condizione di GameOver* quando esaurisce le vite.
- **Interfaccia utente:** il sistema deve fornire un'interfaccia che mostri in tempo reale il punteggio e le vite rimanenti.
- **Menù principale:** il sistema deve offrire un menù iniziale con opzioni per avviare una nuova partita e visualizzare i comandi di gioco.

Requisiti non funzionali

- Il sistema deve garantire una risposta fluida e immediata ai comandi del giocatore.
- Il software deve essere eseguibile su diverse piattaforme desktop, tra cui Windows, macOS e Linux.
- Il sistema deve adattarsi correttamente a diverse risoluzioni video standard, mantenendo proporzioni e leggibilità.

1.2 Modello del Dominio

In questa sezione si descrive il modello del *dominio applicativo*, descrivendo le *entità* in gioco ed i rapporti fra loro. Si possono sollevare eventuali aspetti particolarmente impegnativi, descrivendo perché lo sono, senza inserire idee circa possibili soluzioni, ovvero sull'organizzazione interna del software. Infatti, la fase di analisi va effettuata **prima** del progetto: né il progetto né il software esistono nel momento in cui si effettua l'analisi. La discussione di aspetti propri del software (ossia, della *soluzione* al problema e non del problema stesso) appartengono alla sfera della progettazione, e vanno discussi successivamente.

È obbligatorio fornire uno schema UML del dominio, che diventerà anche lo scheletro della parte "entity" del modello dell'applicazione, ovvero degli elementi costitutivi del modello (in ottica MVC - Model View Controller): se l'analisi è ben fatta, dovreste ottenere una gerarchia di concetti che rappresentano le entità che compongono il problema da risolvere. Un'analisi ben svolta **prima** di cimentarsi con lo sviluppo rappresenta un notevole aiuto per le fasi successive: è sufficiente descrivere a parole il dominio, quindi estrarre i sostantivi utilizzati, capire il loro ruolo all'interno del problema, le relazioni che intercorrono fra loro, e reificarli in interfacce.

Elementi positivi

- Si modella il dominio in forma di UML.
- Viene descritto accuratamente il modello del dominio.
- Il modello cattura le entità di dominio, le loro caratteristiche principali,
- e le relazioni che intercorrono fra di loro.

Elementi negativi

- Manca una descrizione a parole del modello del dominio.
- Manca una descrizione UML delle entità del dominio e delle relazioni che intercorrono fra loro.
- Vengono presentati elementi di design, o peggio, implementativi.
- Viene mostrato uno schema UML che include elementi implementativi o non utili alla descrizione del dominio, ma volti alla soluzione (non devono vedersi, ad esempio, campi o metodi privati).

Esempio

GLaDOS dovrà essere in grado di accedere ad un'insieme di camere di test. Tale insieme di camere prende il nome di percorso. Ciascuna camera è composta di challenge successivi. GLaDOS è responsabile di associare a ciascun challenge un insieme di consigli (suggestions) destinati all'utente (subject), dipendenti da possibili eventi. GLaDOS dovrà poter comunicare coi locali cucina per approntare le torte. Le torte potranno essere dolci, oppure semplici promesse di dolci che verranno disattese. Gli elementi costitutivi il problema sono sintetizzati in Figura 1.1.

Data la complessità di elaborare consigli via AI senza intervento umano, la prima versione del software fornita prevederà una serie di consigli forniti dall'utente. Il requisito non funzionale riguardante il consumo energetico richiederà studi specifici sulle performance di GLaDOS che non potranno essere effettuati all'interno del monte ore previsto: tale feature sarà oggetto di futuri lavori.

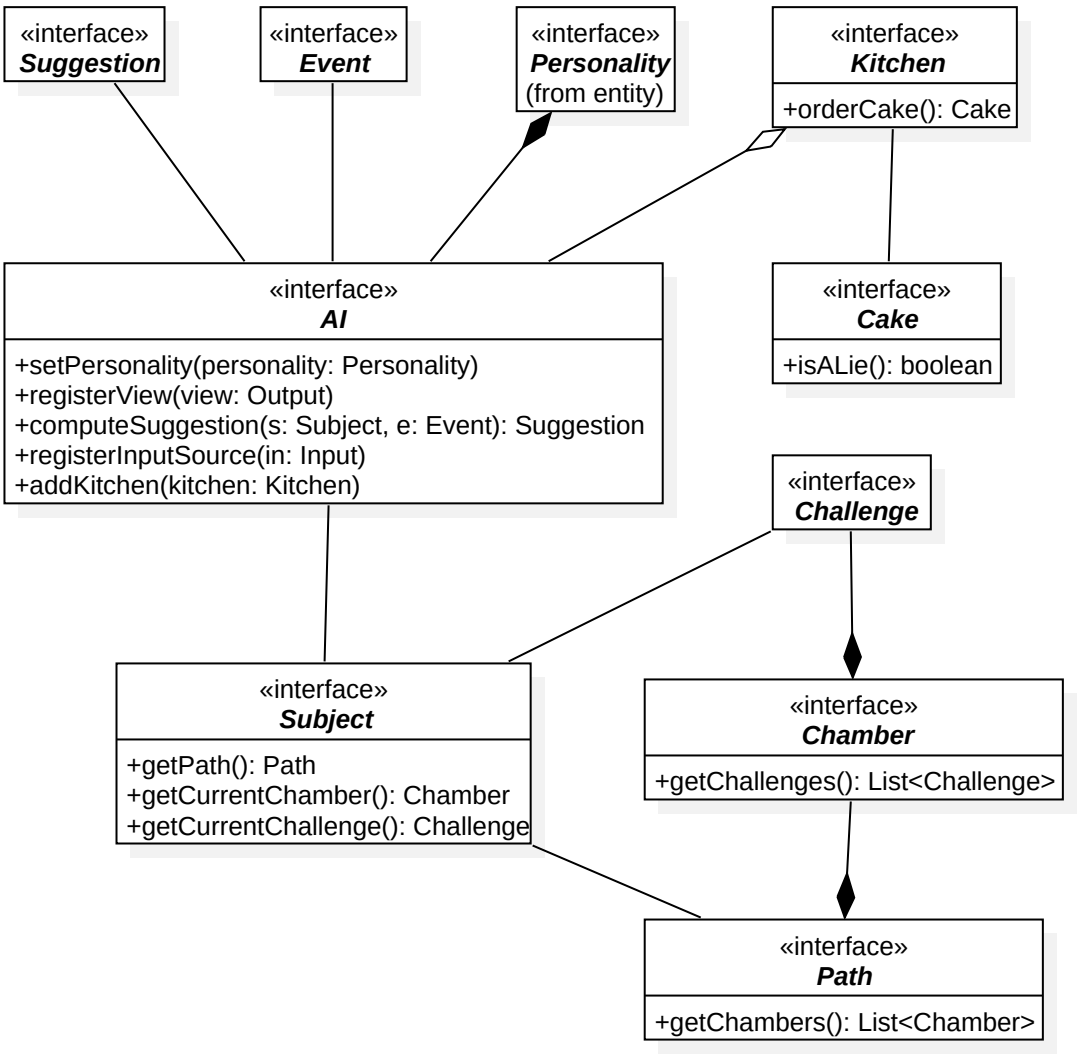

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

In questo capitolo si spiegano le strategie messe in campo per soddisfare i requisiti identificati nell'analisi.

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.

2.1 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 (os-

sia, 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 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.

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.

2.2 Design dettagliato

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

[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.

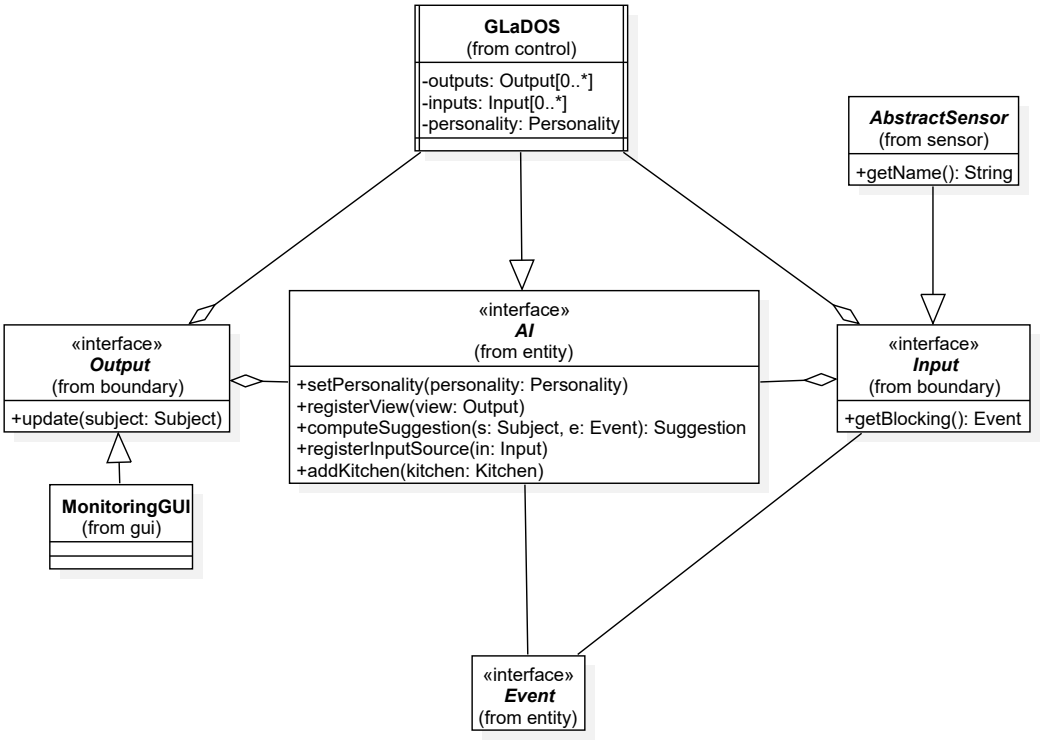

Figura 2.1: Schema UML architetturale di GLaDOS. L'interfaccia `GLaDOS` è il controller del sistema, mentre `Input` ed `Output` sono le interfacce che mappano la view (o, più correttamente in questo specifico esempio, il boundary). Un'eventuale interfaccia grafica interattiva dovrà implementarle entrambe.

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

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 2.2: le implementazioni di `Personality` possono essere modificate, e la modifica impatta direttamente sul comportamento di GLaDOS.

Riuso del codice 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 2.3. Il metodo template è `onSuccess()`, che chiama un metodo astratto e protetto `makeCake()`.

Gestione di output multipli

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 2.4

Contro-esempio: pessimo diagramma UML

In Figura 2.5 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.

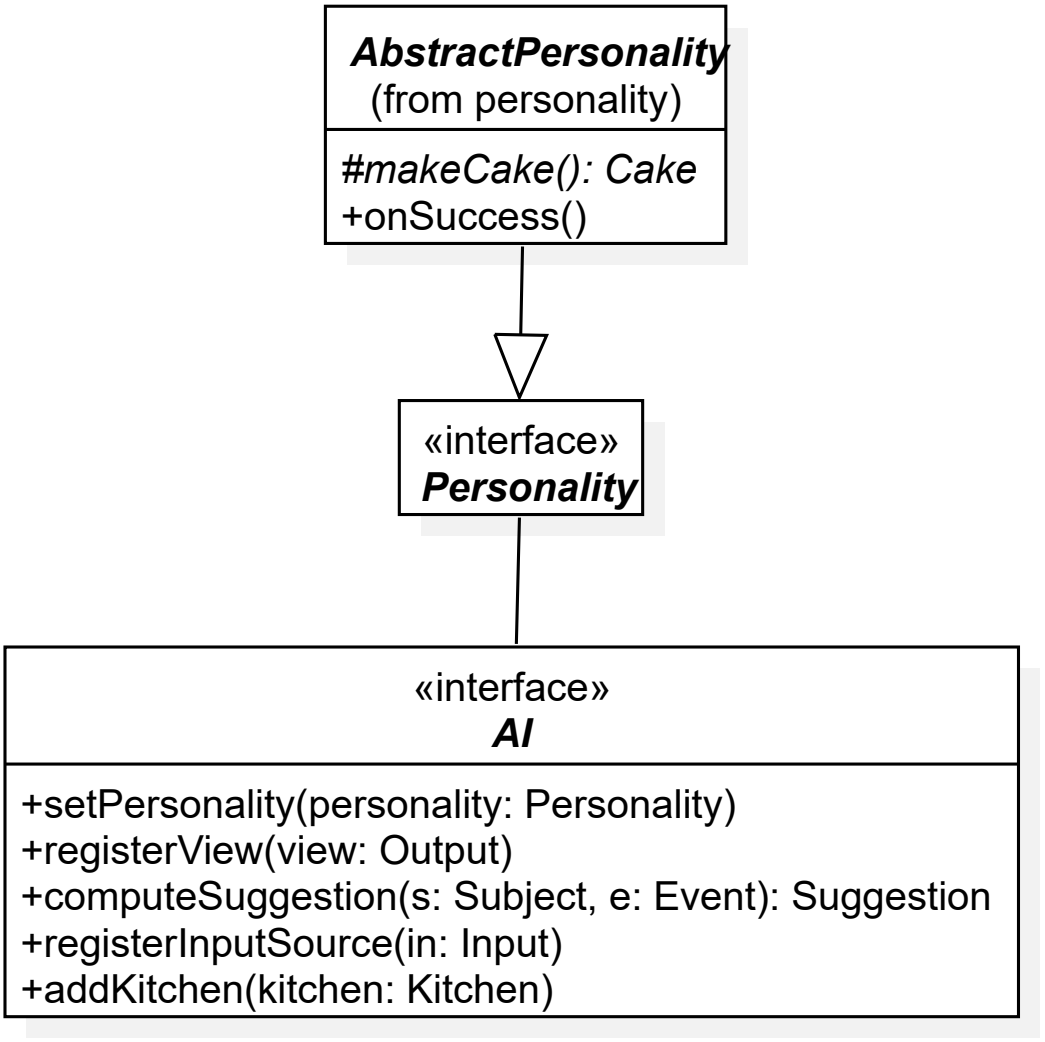

Figura 2.2: Rappresentazione UML del pattern Strategy per la personalità di GLaDOS

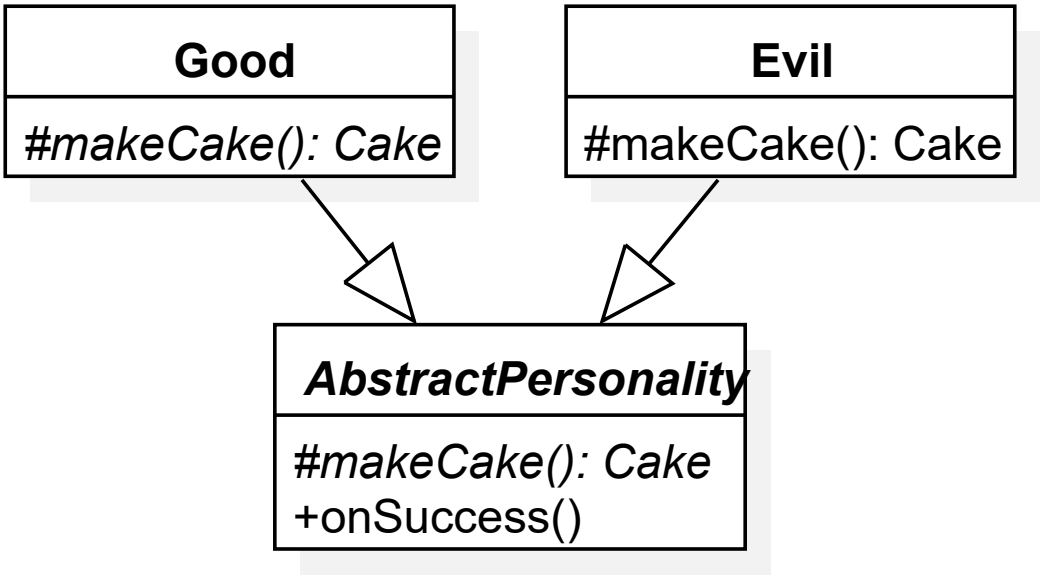

Figura 2.3: Rappresentazione UML dell'applicazione del pattern Template Method alla gerarchia delle Personalità

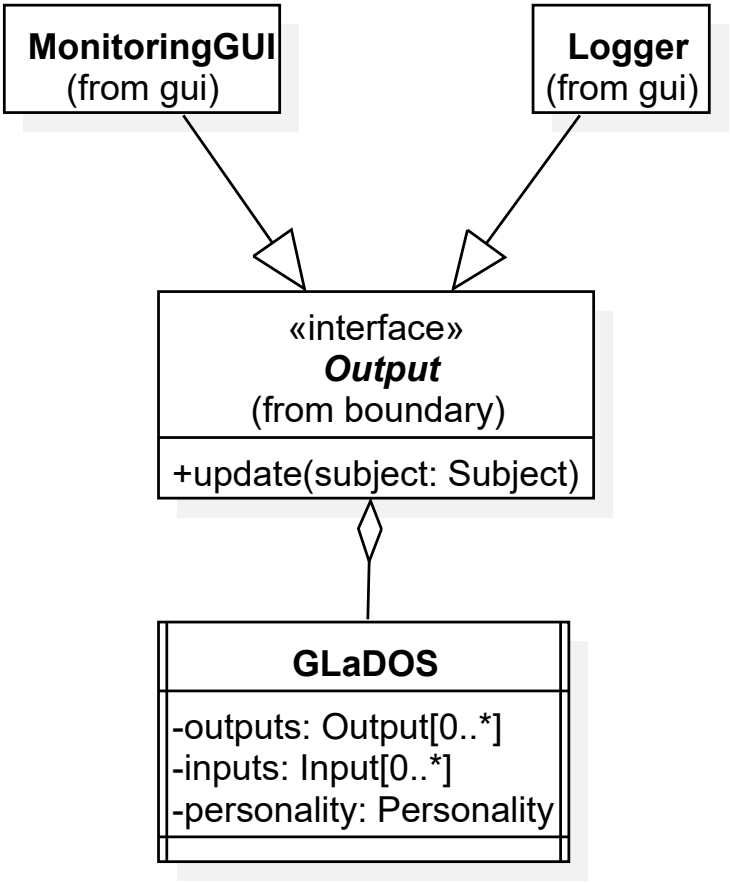

Figura 2.4: Il pattern Observer è usato per consentire a GLaDOS di informare tutti i sistemi di output in ascolto

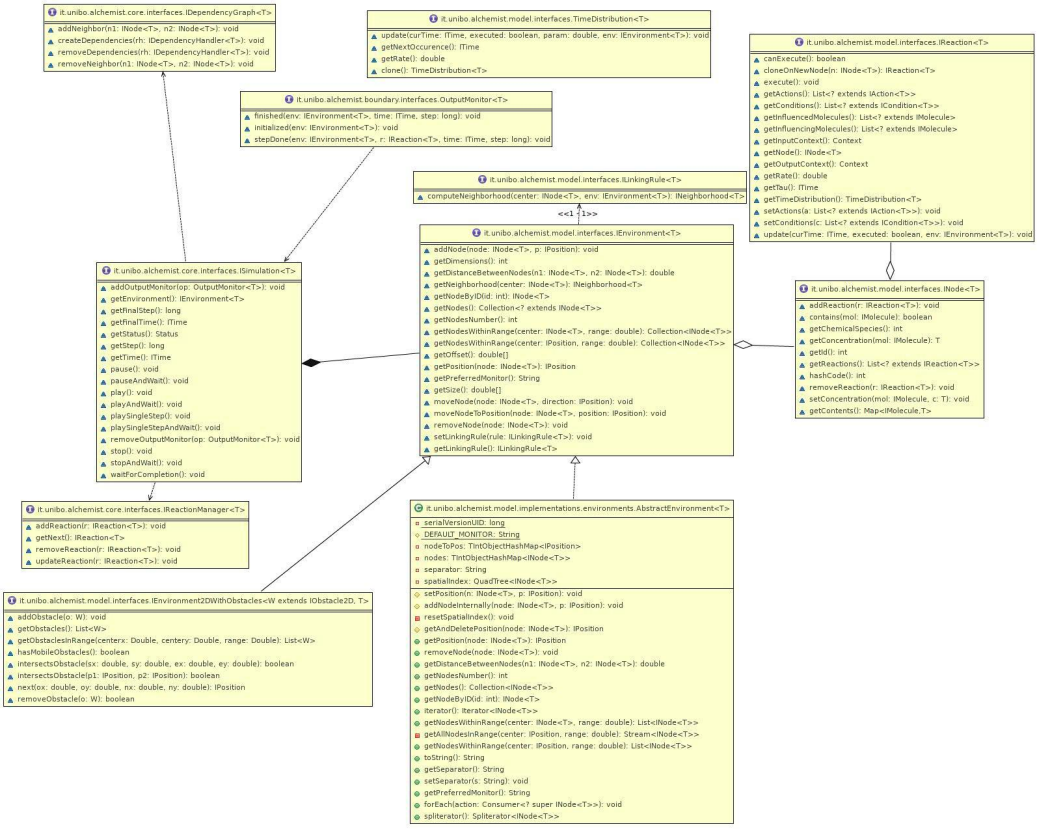

Figura 2.5: Schema UML mal fatto e con una pessima descrizione, che non aiuta a capire. Don't try this at home.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

In questo progetto abbiamo implementato dei test unitari con JUnit 5 per tutte le classi principali. I test progettati assicurano la verifica automatica delle funzionalità fondamentali del softaware. Di seguito vengono riportati alcuni esempi di test implementati.

Common

- `TestBoundingBox`: verifica la corretta creazione e il corretto funzionamento delle dimensioni delle entità.
- `TestResourceLoader`: viene verificato il giusto caricamento delle risorse del gioco.
- `TestPosition`: testa la gestione delle posizioni nello spazio di gioco.
- `TestVector`: verifica le operazioni sui vettori.

Controller

- `TestInputManager`: controlla la gestione degli input da tastiera.
- `TestGameController`: testa la logica principale di controllo del gioco.

Model

- `TestMario`: controlla che il movimento e le funzionalità del personaggio principale avvengano correttamente.
- `TestLivesManager`: controlla la gestione delle vite del personaggio principale.
- `TestNormalState`: verifica il comportamento base del personaggio nello stato normale.
- `TestWithHammerState`: verifica l'attivazione del potere alla raccolta del martello, e la sua durata.
- `TestPlatform`: controlla il comportamento delle piattaforme normal e breackable.
- `TestLadder`: verifica il funzionamento delle scale.
- `TestGameTank`: verifica il funzionamento della tanica d'olio.
- `TestPauline`: testa il comportamento dell'ostaggio.
- `TestDonkeyKong`: verifica la logica del lancio dei barili e il comportamento dell'antagonista.
- `TestBarrel`: controlla il movimento e le collisioni dei barili.
- `TestFire`: verifica il comportamento delle fiamme.
- `TestCollectible`: testa la raccolta degli oggetti collezionabili (monete, martelli) e il loro effetto.
- `TestEntry:` verifica la corretta gestione delle singole voci della classifica.
- `TestGameLeaderBoard:` verifica la gestione e il salvataggio dei punteggi nella leaderboard.
- `TestScore:` controlla la gestione e il calcolo dei punteggi.
- `TestBonus:` verifica la logica dei bonus di gioco.
- `TestInGameModelState:` testa lo stato del modello durante la partita.
- `TestMenuModelState:` verifica il comportamento del modello nel menu principale.
- `TestPauseModelState:` controlla la gestione dello stato di pausa.

Repository

- `TestScoreRepository`: verifica la corretta gestione della persistenza e del recupero dei punteggi nella repository.

3.2 Note di sviluppo

3.2.1 Alessandro Rebosio

Progettazione con `generics`

Utilizzati i generics per definire interfacce e classi parametrizzate. Permalink
`https://github.com/alessandrorebosio/OOP24-coffeBreak/blob/3a264084eabd86bac19f9978d9c2aba0c1cbf624/src/main/java/it/unibo/coffebreak/api/common/State.java#L14`

Utilizzo di `Stream`

Utilizzati di frequente, soprattutto per il controllo sulle entità. Permalink di un esempio
`https://github.com/alessandrorebosio/OOP24-coffeBreak/blob/3a264084eabd86bac19f9978d9c2aba0c1cbf624/src/main/java/it/unibo/coffebreak/impl/model/physics/collision/GameCollision.java#L54-L67`

Utilizzo di lambda expressions

Utilizzati di frequente, nel caricamento delle risorse. Permalink di un esempio:
`https://github.com/alessandrorebosio/OOP24-coffeBreak/blob/3a264084eabd86bac19f9978d9c2aba0c1cbf624/src/main/java/it/unibo/coffebreak/impl/common/ResourceLoader.java#L99`

Gestione degli `Optional`

Usato per gestire valori che potrebbero essere assenti. Permalink di un esempio
`https://github.com/alessandrorebosio/OOP24-coffeBreak/blob/3a264084eabd86bac19f9978d9c2aba0c1cbf624/src/main/java/it/unibo/coffebreak/impl/model/states/ingame/InGameModelState.java#L57-L61`

3.2.2 Istruzioni 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

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 [MN98].
- 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

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

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.

4.2 Difficoltà incontrate e commenti per i docenti

Questa sezione, **opzionale**, può essere utilizzata per segnalare ai docenti eventuali problemi o difficoltà incontrate nel corso o nello svolgimento del

progetto, può essere vista come una seconda possibilità di valutare il corso (dopo quella offerta dalle rilevazioni della didattica) avendo anche conoscenza delle modalità e delle difficoltà collegate all'esame, cosa impossibile da fare usando le valutazioni in aula per ovvie ragioni. È possibile che alcuni dei commenti forniti vengano utilizzati per migliorare il corso in futuro: sebbene non andrà a vostro beneficio, potreste fare un favore ai vostri futuri colleghi. Ovviamente *il contenuto della sezione non impatterà il voto finale.*

Appendice A

Guida utente

A.1 Menu Principale

All'avvio dell'applicazione, l'unico strumento a disposizione sarà la tastiera. Nel menu principale sarà possibile visualizzare la leaderboard e selezionare le opzioni disponibili tramite le frecce `SU` e `GIÙ`. Per confermare la selezione si utilizza il tasto `ENTER`.

A.2 Durante il Gioco

Entrati in gioco, si parte dal primo livello: lo scopo è raggiungere la principessa evitando i nemici e raccogliendo power-up.

Per giocare basta usare la tastiera: ogni `freccia` muove il personaggio nella rispettiva direzione, la `barra spaziatrice` fa saltare, mentre le frecce `SU` e `GIÙ` funzionano solo in corrispondenza di una scala. In qualsiasi momento, premendo il tasto `ESC` puoi mettere il gioco in pausa.

A.3 Menu di Pausa

Nel menu di pausa si naviga tra le opzioni disponibili con le frecce `SU` e `GIÙ`, e si seleziona l'opzione desiderata con `ENTER`.

A.4 Game Over

Quando il personaggio perde tutte le vite, si entra nella schermata di Game Over. In questa schermata l'unica azione possibile è premere `ENTER` per tornare al menu principale.

Appendice B

Esercitazioni di laboratorio

B.1 alessandro.rebosio@studio.unibo.it

- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=177162#p246059
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p248324
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=180101#p249805
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250995

Bibliografia

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.