

Relazione del progetto
”coffeBreak”

Grazia Bochdanovits de Kavna
Alessandro Rebosio
Filippo Riccioti

7 luglio 2025

Indice

1	Analisi	2
1.1	Descrizione e requisiti	2
1.2	Modello del Dominio	4
2	Design	5
2.1	Architettura	5
2.2	Design dettagliato	7
3	Sviluppo	17
3.1	Testing automatizzato	17
3.2	Note di sviluppo	19
3.2.1	Alessandro Rebosio	19
3.2.2	Grazia Bochdanovits de Kavna	20
4	Commenti finali	21
4.1	Autovalutazione e lavori futuri	21
4.2	Difficoltà incontrate e commenti per i docenti	21
A	Guida utente	23
A.1	Menu Principale	23
A.2	Durante il Gioco	23
A.3	Menu di Pausa	23
A.4	Game Over	23
B	Esercitazioni di laboratorio	24
B.1	alessandro.rebosio@studio.unibo.it	24

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 collisioni 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 buone 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

Il gioco si avvia da un **menù iniziale**, che consente al giocatore di avviare una nuova partita, visualizzare i cinque punteggi più alti raggiunti nelle sessioni precedenti oppure uscire dal gioco. Al momento dell'avvio della partita, il gioco carica il **primo livello** della rotazione ispirata a *Donkey Kong* (Nintendo, 1981), e posiziona automaticamente il personaggio principale (Jumpman) nella posizione iniziale in basso a sinistra.

La **partita si sviluppa su due livelli arcade classici**, che si alternano in rotazione ad ogni completamento, ricreando il comportamento dell'originale. Ogni livello presenta una disposizione fissa di piattaforme, scale, ostacoli e nemici (barili o fiamme), con un obiettivo specifico: raggiungere la cima dello schermo evitando gli ostacoli e salvare la damigella in pericolo.

Il personaggio è controllabile tramite input da tastiera e può muoversi lateralmente, salire scale e saltare. Il gioco gestisce le collisioni con gli elementi di gioco (piattaforme, scale, ostacoli e oggetti bonus) permettendo un'esperienza coerente con l'originale.

La visuale è statica: l'intero livello è visibile in una singola schermata. Tuttavia, il personaggio può cadere oltre la parte inferiore dello schermo. In tal caso, o se viene colpito da un ostacolo, si attiva la condizione di **game over** o di perdita di una vita, mostrando il punteggio accumulato nella sessione in corso e confrontandolo con il record precedente.

Durante la partita, il **punteggio** è visibile nella parte superiore dello schermo e si aggiorna dinamicamente in base alle azioni del giocatore (es. salto dei barili, raccolta bonus, tempo residuo). È anche possibile mettere il gioco in **pausa** e riprendere la partita in qualsiasi momento.

Una volta terminata la partita (completamento dei livelli o esaurimento delle vite), viene mostrata una schermata di fine con l'opzione per tornare al menù principale, avviare una nuova partita o uscire.

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

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

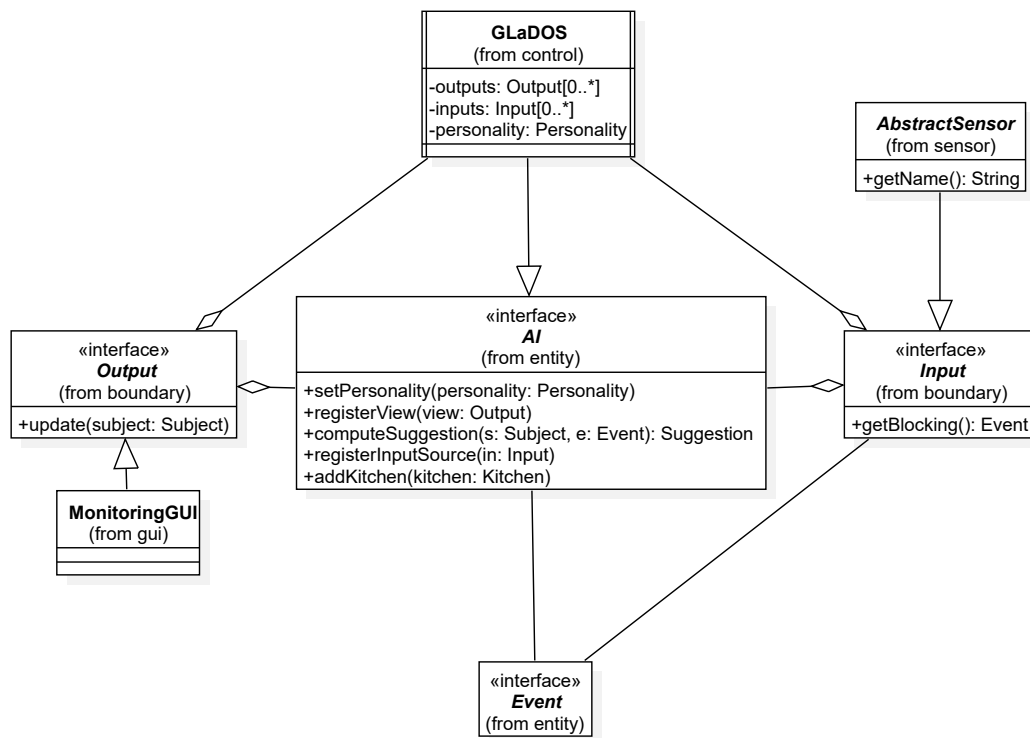


Figura 2.1: Schema UML architetturale di GLaDOS. L'interfaccia **GLaDOS** è il controller del sistema, mentre **Input** ed **Output** sono le interfacce che mappano la view (o, più correttamente in questo specifico esempio, il boundary). Un'eventuale interfaccia grafica interattiva dovrà implementarle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

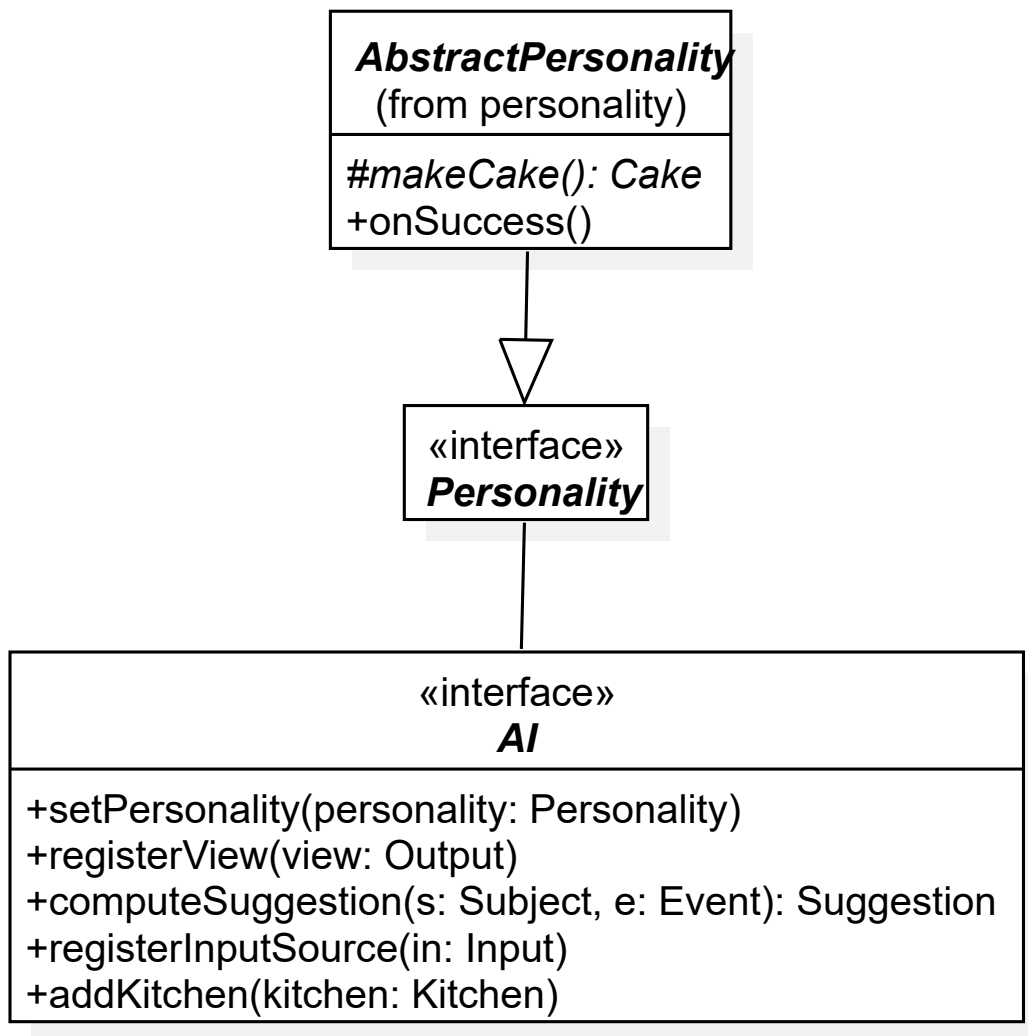


Figura 2.2: Rappresentazione UML del pattern Strategy per la personalità di GLaDOS

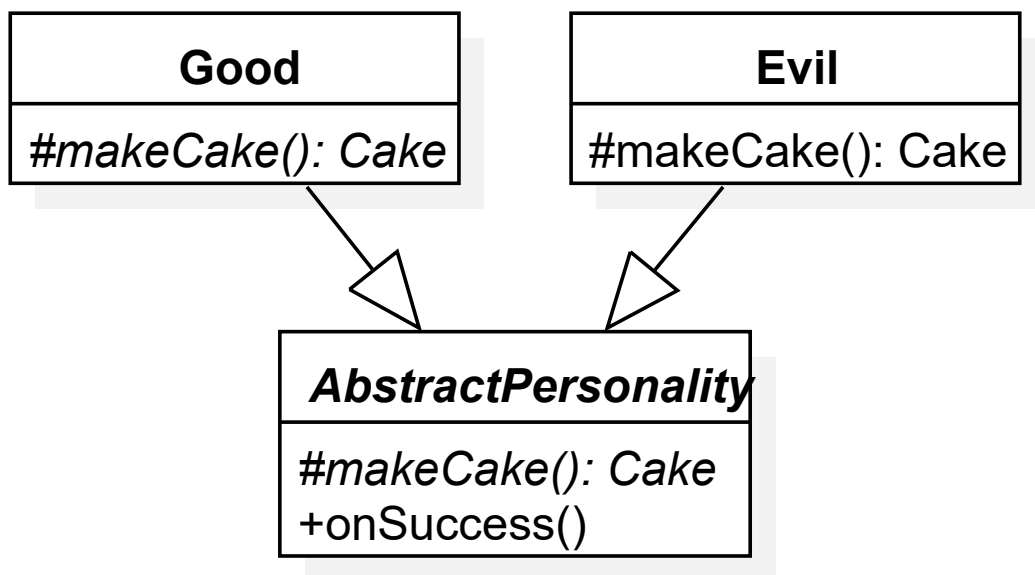


Figura 2.3: Rappresentazione UML dell'applicazione del pattern Template Method alla gerarchia delle Personalità

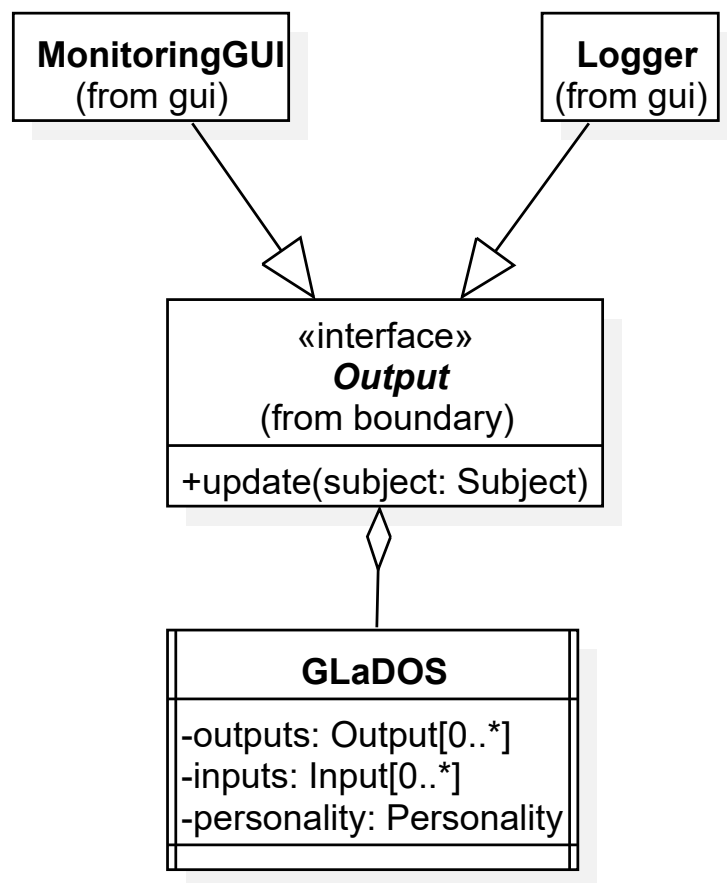


Figura 2.4: Il pattern Observer è usato per consentire a GLaDOS di informare tutti i sistemi di output in ascolto

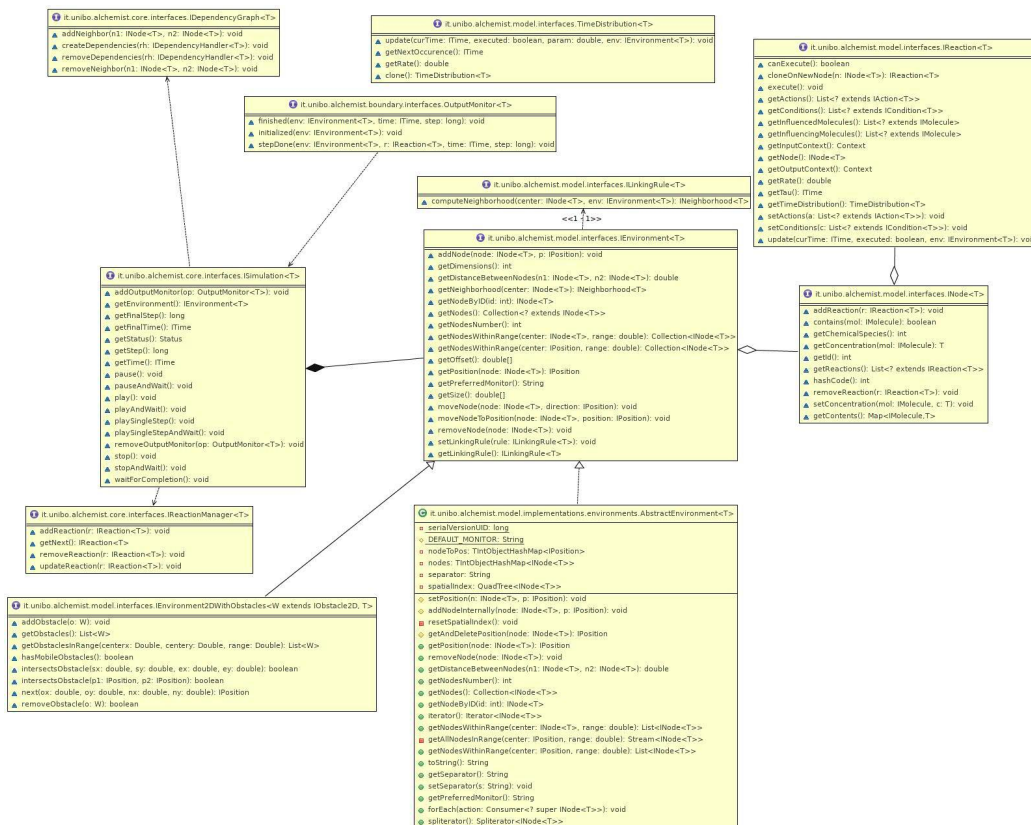


Figura 2.5: Schema UML mal fatto e con una pessima descrizione, che non aiuta a capire. Don't try this at home.

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 software. 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

- **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/00P24-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/00P24-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 immagini. Permalink di un esempio: <https://github.com/alessandrorebosio/00P24-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/00P24-coffeBreak/blob/3a264084eabd86bac19f9978d9c2aba0c1cbf624/src/main/java/it/unibo/coffebreak/impl/model/states/ingame/InGameModelState.java#L57-L61>

3.2.2 Grazia Bochdanovits de Kavna

Utilizzo di Stream

Utilizzati di frequente. Permalink di un esempio

<https://github.com/alessandrorebosio/00P24-coffeBreak/blob/926d3f1b985c9bba79c5f766ac0070b04fd6185f/src/main/java/it/unibo/coffebreak/impl/view/states/ingame/InGameView.java#L62-L65>

Gestione degli Optional

Usato per gestire valori che potrebbero essere assenti. Permalink di un esempio

<https://github.com/alessandrorebosio/00P24-coffeBreak/blob/926d3f1b985c9bba79c5f766ac0070b04fd6185f/src/main/java/it/unibo/coffebreak/impl/view/render/GameRenderManager.java#L82-L87>

Utilizzo di lambda expressions

Utilizzati di frequente, soprattutto nel controllo delle collisioni fra entità. Permalink di un esempio:

<https://github.com/alessandrorebosio/00P24-coffeBreak/blob/926d3f1b985c9bba79c5f766ac0070b04fd6185f/src/main/java/it/unibo/coffebreak/impl/model/entities/mario/Mario.java#L173-L186>

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

- [Cla25] ClassicGaming.cc. Donkey kong - sounds. Online resource, 2025. Archivio sonoro dei suoni originali dell'arcade Donkey Kong (Nintendo, 1981), utilizzato per integrare effetti sonori autentici nel progetto. Include salti, cadute, rumori ambientali e suoni di interazione.
- [Goo25] Google Fonts. Press start 2p — google fonts. Font disponibile tramite Google Fonts, 2025. Font bitmap ispirato alla tipografia dei videogiochi arcade degli anni '80, usato per i testi e i punteggi nel progetto. Garantisce coerenza stilistica con l'estetica retro.
- [The25] The Spriters Resource. Donkey kong (arcade) sprites. Online sprite archive, 2025. Collezione completa degli sprite estratti dal gioco originale Donkey Kong. Utilizzati per ricostruire graficamente il gameplay e l'interfaccia in stile retro. Include animazioni di Mario, Donkey Kong, ostacoli e oggetti.