Relazione di Progetto
"J-Pou"

Diego Andruccioli, Rei Mici, Giovanni Morelli

10 gennaio 2026

Sommario

Questa relazione descrive il progetto "J-Pou", un simulatore di animale domestico virtuale sviluppato in Java con JavaFX per il corso di Programmazione e Sviluppo del Software 2025/26. Il documento presenta l'analisi, la progettazione e l'implementazione di un'applicazione interattiva ispirata ai Tamagotchi di prima generazione e al gioco mobile "Pou", in cui l'utente gestisce il ciclo vitale di una creatura digitale attraverso la cura quotidiana e attività ludiche.

Il progetto è strutturato secondo il pattern architetturale Model-View-Controller, garantendo separazione tra logica di dominio, interfaccia grafica e gestione degli input. Il modello include statistiche vitali che decadono nel tempo, un sistema di inventario e shop per la gestione delle risorse, minigiochi per l'intrattenimento, e un meccanismo di persistenza basato su JSON per salvare lo stato tra sessioni di gioco.

La relazione documenta le scelte progettuali adottate dal team di sviluppo, analizzando non solo le soluzioni tecniche finali ma anche il processo di revisione critica che ha portato al loro consolidamento. Per ogni componente vengono presentati il problema affrontato, le soluzioni considerate, gli schemi UML e i pattern utilizzati, tra cui MVC, Observer e Factory Method. Il documento include inoltre una sezione dedicata al testing automatizzato con JUnit 5 e code coverage tramite JaCoCo.

È presente una sezione finale in cui ogni membro del team sviluppa un'autovalutazione e una riflessione personale rivolta ai professori su come è avvenuto lo sviluppo del progetto, evidenziando le sfide affrontate nella gestione dell'architettura e del tempo. Il documento si conclude con una guida utente che spiega come utilizzare il gioco.

Indice

Capitolo 1

Analisi

In questo capitolo viene affrontata l'analisi del problema alla base del progetto "J-Pou". L'obiettivo è duplice: da un lato, definire in modo esaustivo i requisiti del software, delineando cosa il sistema deve essere in grado di fare, dall'altro descrivere il modello del dominio, definendo le entità virtuali e le logiche che governano l'interazione tra l'utente e il sistema.

1.1 Descrizione del progetto: J-Pou

Il software mira alla realizzazione di un simulatore di animale domestico virtuale (virtual pet) denominato "J-Pou", ispirato al celebre gioco mobile "Pou". Per simulatore di animale domestico virtuale si intende un'applicazione in cui l'utente gestisce il ciclo di vita di una creatura digitale, prendendosene cura attraverso interazioni dirette e attività ludiche.

Il sistema modella un'entità virtuale dotata di bisogni fisiologici ed affettivi che evolvono nel tempo. L'animale possiede caratteristiche vitali (fame, energia, salute, divertimento) che decadono automaticamente, richiedendo l'intervento dell'utente per essere ripristinate. L'utente può interagire con l'animale in diversi modi: nutrendolo con cibo, somministrandogli pozioni curative, facendolo riposare, lavandolo, e intrattenendolo attraverso minigiochi. L'ambiente di gioco è organizzato in stanze tematiche (cucina, camera da letto, bagno, infermeria, sala giochi), ciascuna dedicata a specifiche attività di cura.

Il sistema prevede un'economia interna basata su monete virtuali: l'utente guadagna monete completando il minigioco e le spende in un negozio per acquistare beni di consumo (cibo, pozioni) e oggetti estetici (vestiti). Gli acquisti vengono conservati in un inventario personale e possono essere utilizzati in momenti successivi. L'obiettivo principale è mantenere in vita l'animale il più a lungo possibile, bilanciando le risorse economiche con le necessità vitali. Se le statistiche vitali scendono sotto livelli critici per troppo tempo, l'animale potrebbe morire, terminando la partita.

Requisiti funzionali

- Il sistema dovrà gestire automaticamente il decadimento delle statistiche vitali dell'animale nel tempo, distinguendo tra stato di veglia e stato di riposo.
- L'utente dovrà poter navigare liberamente tra le diverse stanze e interagire contestualmente con l'animale in base all'ambiente in cui si trova.

- Il sistema dovrà fornire un negozio per l'acquisto di oggetti utilizzando monete virtuali, e un inventario per la gestione degli oggetti acquistati.
- Dovrà essere presente almeno un minigioco arcade accessibile dalla sala giochi, necessario per guadagnare monete e intrattenere l'animale.
- Il sistema dovrà garantire la persistenza dello stato di gioco, salvando automaticamente i progressi tra sessioni diverse.

Requisiti non funzionali

- Il sistema dovrà garantire un'esperienza fluida e reattiva, con feedback immediati alle azioni dell'utente.
- L'interfaccia dovrà essere intuitiva e comprensibile senza necessità di consultare manuali o menu complessi.
- Il sistema dovrà essere eseguibile su qualsiasi piattaforma dotata di Java Virtual Machine aggiornata.

1.2 Modello del Dominio

Nel dominio dell'applicazione interagiscono due entità principali: l'**Utente** e il **Pou**. Il Pou è un'entità biologica virtuale che possiede uno stato vitale mutevole nel tempo, caratterizzato da quattro statistiche: Fame, Energia, Salute e Divertimento. Esso interagisce biologicamente con le risorse: consuma cibo, assume pozioni e riposa.

L'Utente agisce come gestore delle risorse: è responsabile dell'acquisizione e della somministrazione dei beni necessari alla sopravvivenza del Pou. L'interazione avviene all'interno di contesti specifici chiamati **Stanze** (Cucina, Infermeria, Camera da Letto, Bagno, Sala Giochi, Shop). Nella Cucina, l'Utente seleziona il **Cibo** dall'inventario e lo somministra al Pou. Nell'Infermeria, l'Utente seleziona le **Pozioni** dall'inventario per ripristinare la salute del Pou. Nella Camera da Letto, l'Utente può indurre il Pou a riposare e accedere al guardaroba per fargli indossare **Vestiti**. Nel Bagno, l'Utente può lavare il Pou. Nella Sala Giochi, l'Utente e il Pou interagiscono tramite **Minigiochi**, che generano Monete e incrementano il Divertimento.

La componente economica lega le due entità: l'Utente utilizza le Monete guadagnate per acquistare Cibo, Pozioni e Vestiti nello **Shop**, rendendoli disponibili nell'Inventario. Gli elementi costitutivi del problema sono sintetizzati in Figura 1.1.

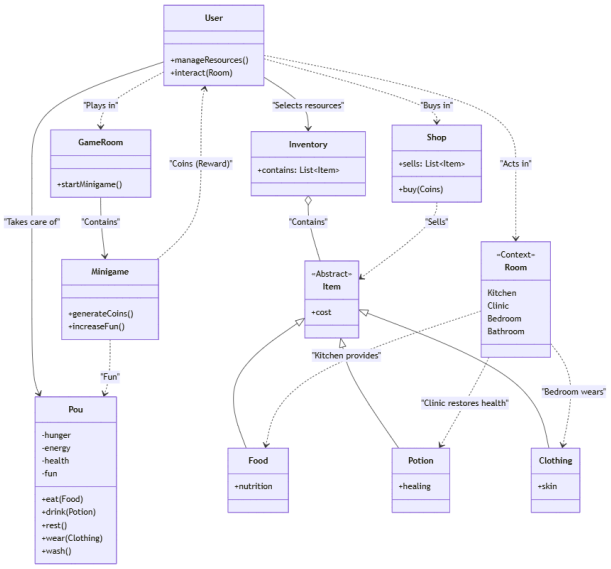

Figura 1.1: Modello del dominio del progetto J-Pou

Capitolo 2

Design

In questo capitolo viene presentata la progettazione del sistema 'J-Pou'. L'obiettivo è duplice: da un lato definire l'architettura generale del software, strutturando l'applicazione secondo il pattern MVC per garantire modularità e ordine, dall'altro approfondire il design dettagliato, descrivendo le soluzioni specifiche e i pattern adottati per risolvere le sfide implementative emerse.

2.1 Architettura

L'architettura del sistema è stata progettata seguendo il pattern architetturale **Model-View-Controller** (MVC). Questa scelta risponde alla necessità di mantenere una rigorosa separazione delle responsabilità tra la logica di dominio (il nucleo dell'applicazione), la rappresentazione grafica e la gestione delle interazioni utente. Tale disaccoppiamento è fondamentale per garantire la modularità del software, facilitare lo sviluppo cooperativo e semplificare la manutenzione futura. La struttura del progetto è organizzata nei seguenti componenti architetturali principali, le cui dipendenze sono illustrate in Figura 2.1:

Model
: Rappresenta il nucleo dell'applicazione, contenente le entità del dominio e l'intera logica di business. Il punto d'ingresso principale è `PouLogic`, una classe che funge da *Facade*, coordinando le logiche specifiche delle stanze (es. `BedroomLogic`, `KitchenLogic`) e gestendo lo stato vitale tramite `PouStatistics`. Il Model include anche `PouGameLoop` (implementazione dell'interfaccia `GameLoop`), un componente che gestisce il flusso temporale generando eventi periodici per applicare il decadimento delle statistiche tramite la logica incapsulata in `PouStatisticsDecay`. Il Model non possiede riferimenti ai componenti grafici, ma espone lo stato attraverso *Properties* osservabili (pattern *JavaFX Beans*), permettendo alla View di legarsi ai dati in modo reattivo.

View
: Gestisce l'interfaccia utente basata su JavaFX, visualizzando lo stato corrente e catturando gli input. La View non mantiene un proprio stato isolato, ma utilizza il meccanismo di *Data Binding* per collegarsi direttamente alle *Properties* esposte dal Model (es. aggiornamento barra della salute in tempo reale). Le view specifiche (estensioni di `AbstractRoomView`) delegano la gestione degli eventi utente (click, trascinamenti) ai rispettivi metodi del Controller, mantenendo una separazione netta tra visualizzazione e logica operativa.

Controller

Agisce da intermediario e orchestratore. Il componente principale, `MainController`, gestisce il ciclo di vita dell'applicazione e il loop di gioco, ma delega compiti specifici a una gerarchia di sotto-controller specializzati (es. `PersistenceController` per il salvataggio, `InventoryController` per la gestione oggetti, e controller dedicati per ogni stanza come `BedroomController`). Questo approccio (Controller Decomposition) evita la creazione di classi monolitiche ("God Class"), distribuendo la responsabilità della gestione degli input in moduli coesi e manutenibili.

Le interazioni seguono un flusso reattivo: la View è legata ai dati del Model tramite Binding; il Controller riceve gli input utente e invoca metodi sul Model (es. `pouLogic.eat()`); il Model aggiorna i suoi valori interni (e verifica vincoli tramite `AbstractStatistic`); le Properties aggiornate propagano automaticamente il cambiamento alla View. Il sistema reagisce parallelamente anche al tempo: il `PouGameLoop` invoca periodicamente l'aggiornamento del Model, scatenando il medesimo flusso di aggiornamento verso l'interfaccia.

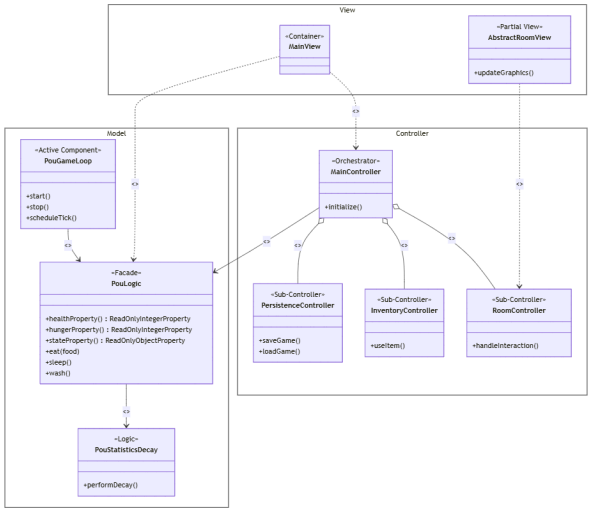

Figura 2.1: Schema architetturale del progetto J-Pou organizzato secondo il pattern MVC.

2.2 Design dettagliato

In questa sezione si approfondiscono gli elementi del design con maggior dettaglio, presentando le soluzioni architetturali adottate per risolvere problemi specifici. Ogni membro del gruppo presenta alcune delle proprie decisioni progettuali seguendo la struttura: problema, soluzione proposta, schema UML, identificazione di pattern utilizzati.

2.2.1 Diego Andruccioli

Validazione delle Statistiche

Problema Le statistiche vitali del Pou (Fame, Energia, Salute, Divertimento) devono essere mantenute rigorosamente nell'intervallo $[0, 100]$ per garantire la coerenza dello stato di gioco. Senza controlli centralizzati, ogni modifica (es. `eat()`, `sleep()`) richiederebbe la ripetizione della logica di validazione, aumentando il rischio di bug e la duplicazione del codice. Per il "portafoglio" (Monete), invece, vale solo il limite inferiore 0 (non sono

ammessi debiti), ma non esiste un limite superiore: riutilizzare la stessa logica delle statistiche vitali sarebbe stato errato.

Soluzione proposta La logica di validazione è stata incapsulata direttamente nel Model tramite il pattern **Template Method**. Il pattern è stato utilizzato per definire lo scheletro dell'algoritmo di gestione del valore in una classe base, lasciando alle sottoclassi solo i dettagli specifici (o l'uso della configurazione di base). Tutte le statistiche vitali estendono la classe astratta `AbstractStatistic`, che implementa l'interfaccia `PouStatistics` e definisce il comportamento comune. Il controllo dei limiti (clamping) è applicato automaticamente nel metodo `setValueStat` della classe madre: in questo modo, si garantisce che l'algoritmo di validazione sia identico e immutabile per tutte le statistiche vitali.

```
public void setValueStat(final int valueStat) {
    final int newValueStat;
    if (valueStat > STATISTIC_MAX_VALUE) {
        newValueStat = STATISTIC_MAX_VALUE;
    } else {
        newValueStat = Math.max(valueStat, STATISTIC_MIN_VALUE);
    }
    this.value.set(newValueStat);
}
```

Listing 2.1: Clamping in AbstractStatistic

Per le monete, la classe dedicata `PouCoins` applica una validazione diversa, poiché non eredita la logica di clamping superiore.

```
public void setCoins(final int value) {
    this.coins.set(Math.max(MIN_COINS, value));
}
```

Listing 2.2: Validazione in PouCoins

Alternative considerate Si è valutato di inserire i controlli nel `PouLogic` o nel `MainController`, ma questo avrebbe violato il principio di responsabilità singola, disperdendo regole di dominio (validità dei dati) al di fuori delle entità stesse.

Schema UML In Figura 2.2 è mostrata la gerarchia delle classi. `AbstractStatistic` centralizza la gestione della `IntegerProperty` e la logica di clamping, mentre le sottoclassi concrete (es. `HungerStatistic`) ereditano automaticamente questo comportamento.

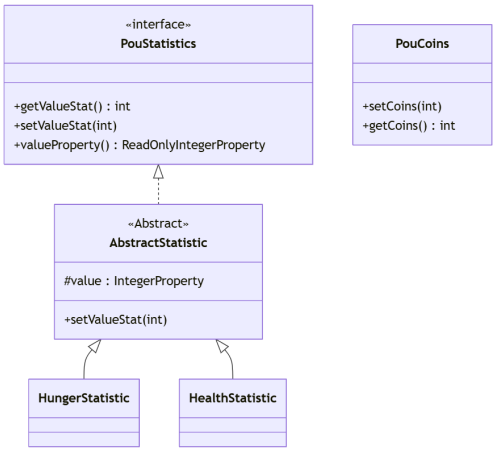

Figura 2.2: Validazione centralizzata

Sistema di Notifica degli Eventi Temporali

Problema Lo stato del Pou deve evolvere automaticamente nel tempo: Fame, Energia, Salute e Divertimento decadono periodicamente indipendentemente dalle azioni dell'utente. Questo processo ("Tick") deve avvenire in background a intervalli regolari, senza bloccare il Main Thread di JavaFX (che gestisce l'interfaccia grafica), per evitare freeze dell'applicazione; inoltre, il componente che scandisce il tempo (`GameLoop`) non deve conoscere la logica di dominio (chi deve essere aggiornato) per mantenere un basso accoppiamento.

Soluzione proposta È stata definita un'interfaccia `GameLoop` che espone il metodo `start()`, `shutdown()` e soprattutto un meccanismo di registrazione eventi basato sul pattern **Observer**. L'utilizzo di questo pattern è stato fondamentale per disaccoppiare la sorgente dell'evento temporale (il `GameLoop`, che funge da **Subject**) dalla logica di reazione (il Controller/Model, che fungono da **Observers**). Il `PouGameLoop` non deve sapere *cosa* succede al tick, ma solo che deve notificare chi è in ascolto. L'implementazione concreta, `PouGameLoop`, utilizza internamente un `ScheduledExecutorService` per generare un tick ogni 30 secondi su un thread separato. Nel `MainControllerImpl` (orchestratore), viene registrata una lambda expression come listener. Questa lambda agisce come concreto Observer: riceve la notifica e invoca la logica di decadimento sul Model e, successivamente, richiede l'aggiornamento della UI tramite `Platform.runLater()`, garantendo la thread-safety.

```
    private void setupGameLoop() {
        if (this.gameLoop instanceof PouGameLoop) {
            ((PouGameLoop) this.gameLoop).addTickListener(() -> {
                // Esegue la logica di dominio (sul thread del timer)
                this.model.applyDecay();
                // Aggiorna la UI (sul thread JavaFX)
                Platform.runLater(this::updateGlobalStatistics);
            });
        }
    }
```

Listing 2.3: Registrazione del listener (Observer) nel MainControllerImpl

In questo modo `PouGameLoop` rimane completamente disaccoppiato: non conosce né `PouLogic` né `MainController`, si limita a "calcolare il tempo" e notificare chiunque sia in ascolto.

Alternative considerate Si è valutato di inserire la logica del loop direttamente nel Controller o nella View. Questo però avrebbe violato il principio di "singola responsabilità" e legato la velocità di gioco al framerate del rendering del minigioco, sprecando risorse per un evento che deve accadere solo due volte al minuto. L'uso di `Observer` (o listener) permette invece di separare nettamente la sorgente dell'evento (Tempo) dal gestore dell'evento (Logica).

Schema UML La struttura segue un pattern Observer semplificato. `PouGameLoop` (Subject) notifica i listener registrati. `MainControllerImpl` (Observer) definisce la reazione all'evento, orchestrando la chiamata al `PouLogic` (Model) e alla `MainView` (UI).

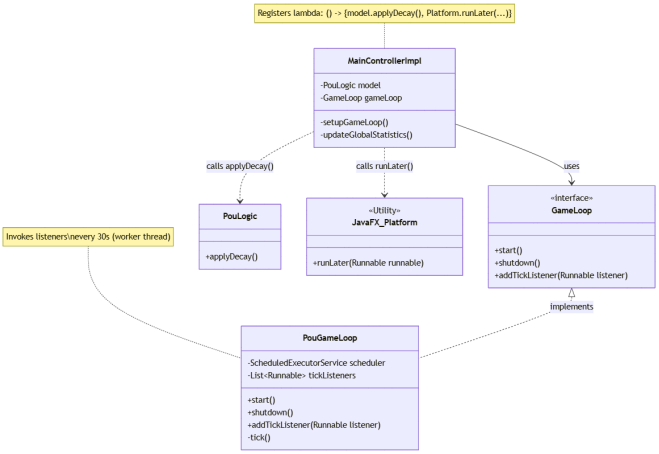

Figura 2.3: Sistema di notifica degli eventi temporali

Persistenza degli Oggetti e Inventario

Problema L'inventario del Pou contiene oggetti eterogenei: Consumabili (Food, Potion) che hanno una quantità, e Durevoli (Skin) che sono unici (sbloccati o no). Serializzare direttamente la mappa Map<Item, Integer> o List<Item> è problematico perché JSON non supporta il polimorfismo nativamente e non conserva il tipo specifico dell'oggetto (es. distinguere una Apple da una HealthPotion solo dai campi). Inoltre, salvare l'intero oggetto (con Nome, Prezzo, Effetto) è ridondante: basta sapere il nome dell'oggetto e la quantità posseduta.

Soluzione proposta È stato adottato il pattern **Data Transfer Object (DTO)** combinato con il pattern **Factory Method**. Il pattern **DTO** è necessario per disaccoppiare il modello di dominio (ricco di logica e comportamenti) dal modello di persistenza (che deve essere semplice e serializzabile). Lo stato dell'inventario viene "appiattito" in due liste separate nel record `SavedInventory` (il DTO), che contiene solo dati puri:

- `items`: lista di record `SavedItem` (coppie nome-quantità) per i consumabili.
- `unlockedSkins`: lista di stringhe (nomi) per le skin sbloccate.

- `equippedSkin`: stringa per la skin correntemente equipaggiata.

Durante il salvataggio in `PersistenceControllerImpl`, l'inventario viene mappato in questi DTO:

```
    /* saving consumable data */
    final List<SavedItem> savedItems = new ArrayList<>();
    this.inventory.getConsumables().forEach((consumable, quantity) ->
            savedItems.add(new SavedItem(consumable.getName(), quantity)
                ↪ )
    );
    /* saving durable data */
    final List<String> unlockedSkins = new ArrayList<>();
    this.inventory.getDurables().forEach(durable ->
            unlockedSkins.add(durable.getName())
    );
```

Listing 2.4: Serializzazione dell'inventario in DTO

Durante il caricamento, per ricostruire gli oggetti complessi dai dati grezzi, è stato utilizzato il pattern **Factory Method** (nella sua variante "Simple Factory"). Il metodo privato `createItemFromName()` agisce come una factory: incapsula la logica di creazione degli oggetti, nascondendo la complessità dell'istanziazione (es. quale classe concreta corrisponde alla stringa "RedSkin") al resto del sistema. Questo centralizza la dipendenza verso le classi concrete in un unico punto, facilitando l'aggiunta di nuovi oggetti in futuro.

```
    private Item createItemFromName(final String name) {
        return switch (name) {
            case Apple.APPLE_NAME -> new Apple();
            case Sushi.SUSHI_NAME -> new Sushi();
            case EnergyPotion.POTION_NAME -> new EnergyPotion();
            case HealthPotion.POTION_NAME -> new HealthPotion();
            case DefaultSkin.DEFAULT_NAME -> new DefaultSkin();
            case GreenSkin.SKIN_NAME -> new GreenSkin();
            case RedSkin.SKIN_NAME -> new RedSkin();
            default -> null;
        };
    }
```

Listing 2.5: Ricostruzione degli oggetti tramite Factory Method

Questo metodo viene invocato ciclicamente per ripopolare l'inventario al momento del caricamento:

```
    /* Inventory recovery */
    for (final SavedItem savedItem : data.inventory().items()) {
        final Item item = createItemFromName(savedItem.name());
        for (int i = 0; i < savedItem.quantity(); i++) {
            this.inventory.addItem(item);
        }
    }
```

Listing 2.6: Ripopolamento dell'inventario

Schema UML La Figura 2.4 mostra come il PersistenceControllerImpl agisca da mappatore tra il dominio (Inventory, Item) e i dati salvati (SavedInventory), utilizzando la Factory interna per l'istanziazione.

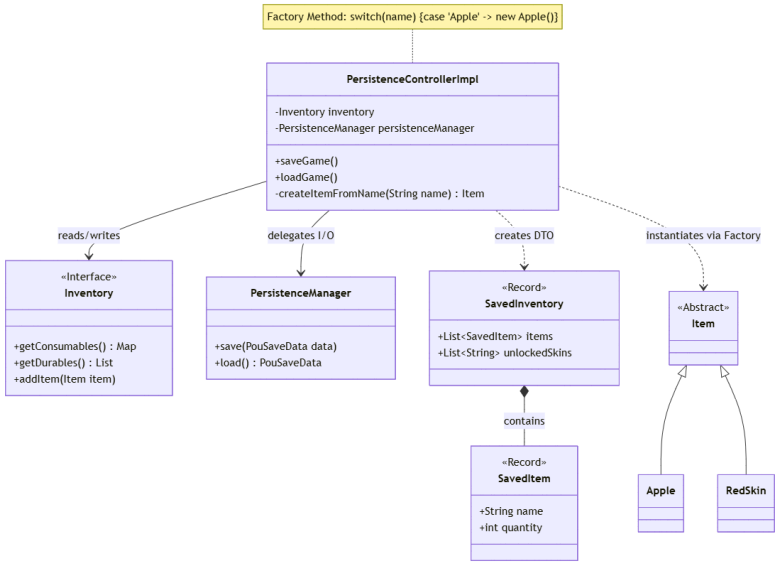

Figura 2.4: Persistenza dell'Inventario. Il controller converte gli Item in DTO leggeri e usa una Factory interna per ricostruirli al caricamento.

Gestione degli Stati e Transizioni

Problema Il Pou non può fare sempre tutto: se dorme non può mangiare, e se muore il gioco deve fermarsi. Gestire questi controlli con semplici variabili boolean (es. `isSleeping`, `isDead`) sparsi nel codice avrebbe portato a confusione e problemi. Serviva un modo centralizzato per definire in che stato si trova il Pou e come questo influenza le regole del gioco (decadimento statistiche) e le interazioni permesse.

Soluzione proposta Ho implementato una versione semplificata del pattern **State**: invece di creare classi separate per ogni stato ho utilizzato un `Enum` (`PouState`) per rappresentare gli stati e uno `switch` nel contesto (`PouLogic/DecayLogic`) per variare il comportamento. Questo approccio mantiene i benefici del pattern State (comportamento che cambia in base allo stato interno) riducendo però la complessità strutturale; inoltre, lo stato cambia radicalmente il funzionamento del "motore temporale":

- Se `AWAKE`: Fame e Divertimento scendono, Salute ed Energia calano piano.
- Se `SLEEPING`: Fame e Divertimento scendono molto piano, ma Salute ed Energia risalgono (rigenerazione).

Questo è implementato con un `switch` nel metodo `applyDecay()`, che delega a strategie diverse:

```
public void applyDecay(final PouStatistics statistics,      final
    ↪ PouState state) {
        switch (state) {
            case AWAKE -> applyAwakeDecay(statistics); case SLEEPING ->
                ↪ applySleepingDecay(statistics);
            case DEAD -> { /* Nessun decadimento se morto */ }
        }
    }
```

Listing 2.7: Logica di decadimento (Strategia) differenziata in base allo Stato

Un aspetto interessante è l'interazione tra View e Stato: nel `BedroomController`, osservo la proprietà dello stato. Quando cambia in `SLEEPING`, applico una classe CSS "night-mode" alla stanza per dare un feedback visivo immediato.

Alternative considerate Avrei potuto usare una classe per ogni stato, ma per soli 3 stati mi sembrava troppo complesso: un Enum con uno switch gestito nel Logic è risultato più semplice da implementare.

Schema UML La Figura 2.5 mostra l'Enum `PouState` usato sia dal `PouLogic` per i controlli, sia da `PouStatisticsDecay` per calcolare i nuovi valori.

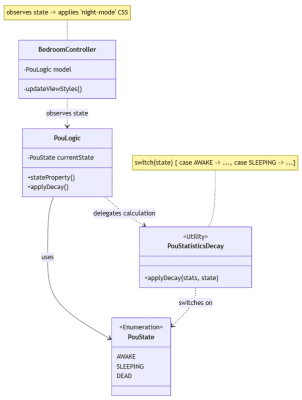

Figura 2.5: Gestione dello Stato. `PouState` centralizza le condizioni del personaggio.

Gestione degli Overlay

Problema Il gioco ha bisogno di schermate che si sovrappongono alla partita (Pausa, Game Over) bloccando le interazioni sottostanti. Non sono semplici finestre ma vere e proprie View personalizzate con bottoni e logica. Inoltre, quando apro la 'Pausa', il tempo di gioco (`GameLoop`) deve fermarsi, altrimenti il Pou continuerebbe ad avere fame mentre l'utente non c'è.

Soluzione proposta L'architettura segue il pattern **Model-View-Controller**. Ogni overlay ha il proprio Controller dedicato (`PauseController`, `GameOverController`) e la propria View, separando nettamente la logica di controllo dalla presentazione. Graficamente, ho utilizzato uno `StackPane` di JavaFX come radice (root) della `MainView`, implementando un approccio a livelli (**Layered View**). Questo layout permette di impilare le viste una sopra l'altra: il gioco normale sta 'sotto', mentre gli overlay vengono caricati e aggiunti 'sopra' nello stack solo quando necessario, intercettando l'input utente. Quando attivo la pausa:

1. Il `MainController` ferma il loop temporale: `gameLoop.stop()`.
2. Mostra la view di pausa sopra il gioco.

Quando riprendo:

1. Nascondo la view di pausa.
2. Riavvio il loop: `gameLoop.start()`.

```
    @Override
    public void pauseGame() {
        if (this.gameLoop.isRunning()) {
            this.gameLoop.stop();
            this.view.showPauseOverlay(true);
        }
    }
    @Override
    public void resumeGame() {
         this.view.showPauseOverlay(false);
         this.gameLoop.start();
    }
```

Listing 2.8: Gestione Pausa nel MainController

Il Game Over è simile, ma irreversibile: offre solo 'Quit' (che resetta il Model) per iniziare una nuova partita.

Schema UML La Figura 2.6 mostra come il `MainController` gestisce la visibilità degli overlay interagendo con i controller specifici e il GameLoop.

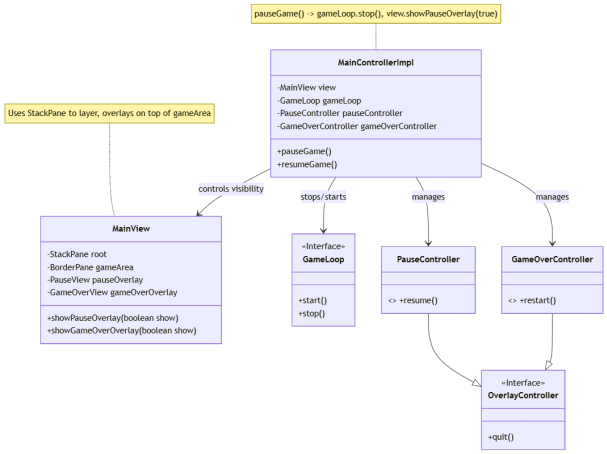

Figura 2.6: Architettura degli Overlay. Sovrapposizione grafica e controllo del flusso temporale.

2.2.2 Rei Mici

Gestione Polimorfica dell'Inventario

Descrizione del problema

La gestione dell'inventario pone una sfida architetturale legata all'eterogeneità delle entità: il sistema deve supportare oggetti consumabili (cibo, pozioni) che richiedono una logica di accumulo, e oggetti durevoli (skin, cosmetici) che sono soggetti a vincoli di unicità (non ha senso possedere due volte la stessa skin).

Un approccio ingenuo basato sull'utilizzo di un'unica `List<Item>` generica costringerebbe a una continua verifica dei tipi a runtime (`instanceof`) e a una gestione procedurale e fragile dei duplicati. Tale soluzione violerebbe il *Single Responsibility Principle*, delegando alla logica di controllo compiti che dovrebbero essere naturalmente gestiti dalla

struttura dati, e rendendo il codice tendente a errori logici, come il decremento della quantità di una skin o la duplicazione di un oggetto unico.

Soluzione proposta

Si è optato per una separazione strutturale e semantica. La gerarchia degli oggetti non è piatta ma si biforca in due sotto-interfacce principali: `Consumable` e `Durable`.

Nel modello concreto `InventoryImpl`, questa distinzione si riflette nell'adozione di strutture dati specializzate che impongono vincoli forti:

- una `Map<Consumable, Integer>` gestisce i consumabili, mappando l'oggetto alla sua quantità e risolvendo nativamente il problema dello *stacking*;
- un `Set<Durable>` gestisce gli oggetti durevoli, garantendo l'unicità degli elementi senza necessità di controlli condizionali manuali.

Schema UML

Lo schema UML di riferimento, riportato in Figura 2.7, evidenzia l'interfaccia radice `Item`, specializzata nelle sotto-interfacce `Consumable` e `Durable`. Le classi concrete (ad esempio `Apple` e `HealthPotion`) implementano `Consumable`, mentre le varianti estetiche (ad esempio `RedSkin`) implementano `Durable`.

La classe `InventoryImpl` non presenta una singola lista, ma relazioni di aggregazione verso le interfacce specifiche, tramite una `Map` di `Consumable` e un `Set` di `Durable`.

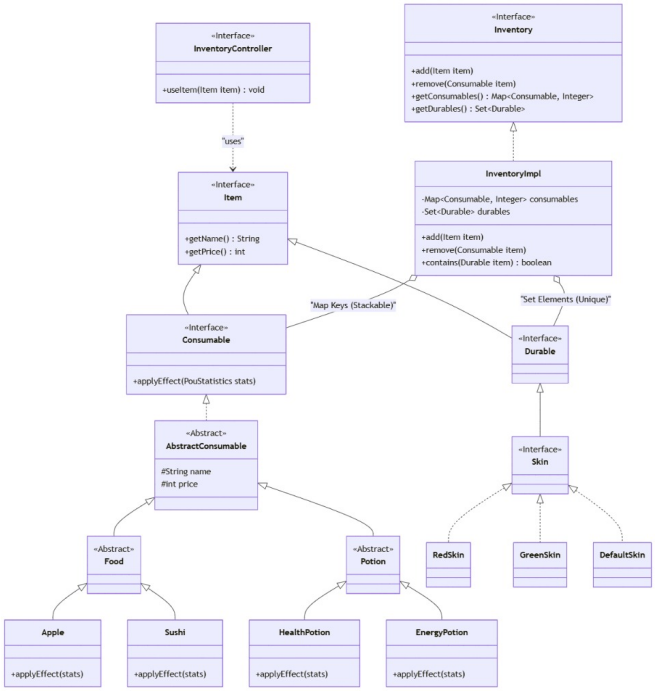

Figura 2.7: Schema UML della gestione polimorfica dell'inventario

Pattern utilizzati

Interface Segregation Principle (SOLID) Invece di forzare tutti gli oggetti ad aderire a un'unica interfaccia che preveda indistintamente metodi di consumo ed equipaggiamento, il sistema applica il principio di *Interface Segregation Principle*. Le interfacce

`Consumable` e `Durable` separano le responsabilità: un oggetto durevole non è tenuto a implementare logiche legate alla quantità o al consumo, e viceversa. Questa scelta aumenta la coesione del codice e facilita l'estensione futura del sistema, ad esempio introducendo nuovi tipi di oggetti non consumabili.

Architectural Pattern: Model-View-Controller (MVC) L'interfaccia `InventoryController` (e la relativa implementazione `InventoryControllerImpl`) agisce come mediatore tra la View e il Modello, che presenta una struttura interna complessa. Anche se il Modello utilizzi tipi specifici, il Controller espone un punto di ingresso polimorfico tramite il metodo `useItem(Item item)`. Tale metodo nasconde la complessità operativa per la View: l'interfaccia grafica non deve conoscere se stia gestendo una mela o una skin, ma si limita a inviare il comando di utilizzo.

È responsabilità del Controller, all'interno del metodo `useItem`, risolvere il tipo dinamico dell'oggetto e delegare l'operazione corretta alla logica di dominio (`PouLogic`), invocando l'azione appropriata. In questo modo, la View rimane completamente agnostica rispetto alla logica di business, preservando il disaccoppiamento imposto dal pattern MVC.

Accesso Contestuale e Proiezione Semantica nelle Viste

Descrizione del problema

In un'architettura in cui l'inventario agisce come *Single Source of Truth*, emerge un conflitto strutturale tra la natura monolitica dei dati e la natura contestuale delle *View*. L'inventario contiene l'universo completo degli oggetti posseduti (cibo, pozioni, skin), mentre le singole stanze (`Bedroom`, `Kitchen`) rappresentano contesti ristretti e specializzati.

Se una `KitchenView` accedesse ai dati dell'inventario senza alcuna forma di filtraggio, l'utente potrebbe visualizzare (e potenzialmente tentare di utilizzare) una *skin* all'interno del frigorifero. Questo, definibile come *inquinamento informativo*, degrada l'esperienza utente ed espone il sistema a stati inconsistenti, costringendo i controller a introdurre controlli difensivi ridondanti.

Soluzione proposta

È stato adottato il pattern della *Proiezione Contestuale dei Dati*. L'alternativa di creare inventari fisicamente separati per ogni stanza (ad esempio `KitchenInventory` o `BedroomInventory`) è stata scartata, poiché avrebbe introdotto duplicazione dei dati e complessi problemi di sincronizzazione.

La soluzione scelta delega alla *View* la responsabilità di agire come *lente semantica* su un unico modello condiviso. Ogni vista applica una catena di filtri ben definiti:

- **Filtro strutturale (backend):** l'interfaccia `Inventory` espone metodi specializzati (`getDurables()`, `getConsumables()`). La `BedroomView` utilizza questo filtro per escludere a priori l'intera categoria dei consumabili.
- **Filtro funzionale (frontend):** la `KitchenView` e la `InfirmaryView`, pur condividendo la sorgente dei consumabili, raffinano ulteriormente il dataset applicando filtri di tipo (`instanceof`) tramite la *Java Stream API*.

Questo favorisce l'estendibilità del sistema: l'aggiunta di una nuova stanza (ad esempio un `Garden`) non richiede modifiche alla struttura dell'inventario, ma esclusivamente la definizione del filtro di visualizzazione appropriato nella nuova *View*.

Schema UML

Il diagramma UML di riferimento, riportato in Figura 2.8, mostra le tre classi *View* (`BedroomView`, `KitchenView`, `InfirmaryView`) che dipendono dalla medesima interfaccia `Inventory`. Le relazioni di dipendenza sono annotate con i vincoli di proiezione semantica: la `BedroomView` è collegata esclusivamente all'interfaccia `Durable`, mentre `KitchenView` e `InfirmaryView` applicano filtri sulle sottoclassi di `Consumable` (`Food` e `Potion`), evidenziando come ciascuna vista osservi porzioni differenti dello stesso modello condiviso.

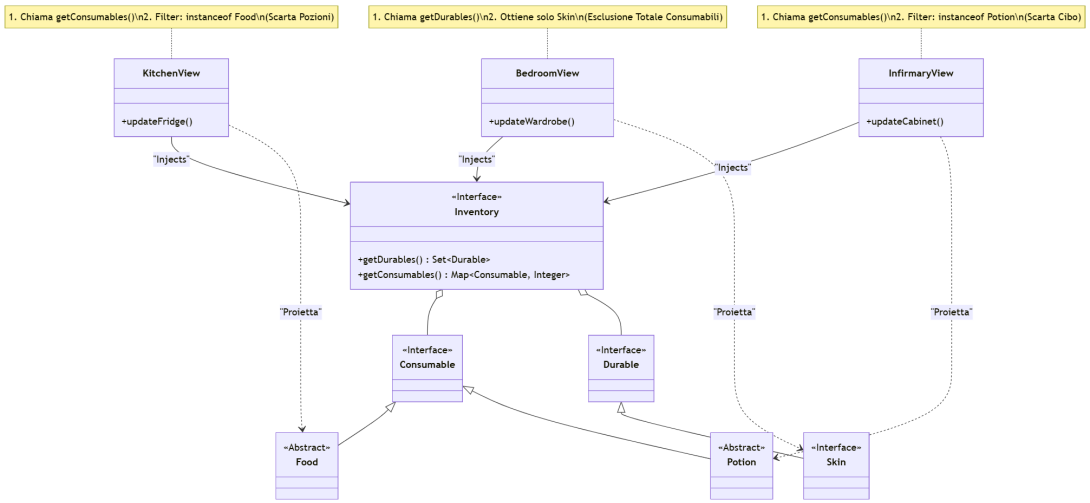

Figura 2.8: Schema UML dell'accesso contestuale e della proiezione semantica nelle viste

Pattern e principi utilizzati

Internal Iterator (via Stream API) Invece di usare il pattern *Iterator* classico (esterno) basato su cicli `for` imperativi, il filtraggio dei dati è realizzato tramite il pattern *Internal Iterator*, implementato nativamente dalle *Stream API* di Java attraverso l'uso di `stream().filter()`. Questo approccio sposta il controllo dell'iterazione dal client alla libreria, consentendo di esprimere la logica di selezione in modo dichiarativo: viene definito *cosa* si vuole ottenere (ad esempio `item instanceof Food`) e non *come* ottenerlo.

Separation of Concerns (SoC) e Information Hiding L'architettura applica la separazione delle resposabilità. Ogni vista non deve conoscere né gestire l'intero insieme degli oggetti di gioco, ma opera esclusivamente sul sottoinsieme rilevante per il proprio contesto funzionale. Attraverso la limitazione della visibilità ai soli oggetti necessari (ad esempio il cibo nella Cucina), si riduce l'accoppiamento tra le componenti. In questo

modo, l'eventuale introduzione futura di nuove categorie di oggetti non impatterebbe sulle viste esistenti, che rimarrebbero immuni al cambiamento poiché il loro meccanismo di filtraggio escluderebbe automaticamente le nuove entità.

Creazione estensibile del catalogo: *Shop Factory*

Descrizione del problema

Il popolamento del negozio (`Shop`) potrebbe avere possibili variazioni, quali ribilanciamento dei prezzi o introduzione di nuovi item. Un approccio ingenuo consiste nell'istanziare direttamente gli oggetti all'interno del costruttore di `ShopImpl` (ad esempio `this.items.add(new Apple())`).

Tale soluzione viola il *Principio Open/Closed* (OCP) e introduce un accoppiamento forte tra la logica del negozio e le classi concrete degli oggetti. Inoltre, questo approccio compromette la testabilità: se il negozio istanzia internamente l'intero catalogo, risulta impossibile testare la logica di acquisto in isolamento, ad esempio utilizzando un inventario ridotto o controllato tramite oggetti *mock*.

Soluzione proposta

È stata introdotta una classe dedicata esclusivamente alla *creation logic*, denominata `ShopFactory`. L'alternativa di adottare il pattern *Singleton* per il negozio, spesso utilizzata in contesti analoghi, è stata deliberatamente scartata al fine di evitare uno stato globale condiviso, che avrebbe limitato la testabilità parallela e l'estendibilità futura del sistema (ad esempio la gestione di più partite o di negozi differenziati).

La soluzione adottata sfrutta il principio della *Dependency Injection*. La classe `ShopImpl` è stata resa passiva e agnostica rispetto al contenuto del catalogo: essa non conosce quali oggetti vende, ma riceve una `Map<Item, Integer>` già popolata tramite il costruttore. In questo modo si ottiene una netta separazione tra la responsabilità di configurazione del catalogo e quella di gestione delle transazioni di acquisto.

Schema UML

Il diagramma UML di riferimento, riportato in Figura 2.9, evidenzia la classe `ShopFactory` come componente esterno al dominio funzionale del negozio. Le relazioni di dipendenza mostrano che:

- `ShopImpl` dipende esclusivamente dall'interfaccia `Item`, senza alcuna conoscenza delle classi concrete come `Apple` o `RedSkin`;
- `ShopFactory` dipende invece dalle classi concrete (`Apple`, `RedSkin`, `Sushi`) che istanzia direttamente;
- `ShopFactory` crea e restituisce l'istanza di `ShopImpl`, invertendo il flusso di controllo della creazione rispetto all'approccio tradizionale.

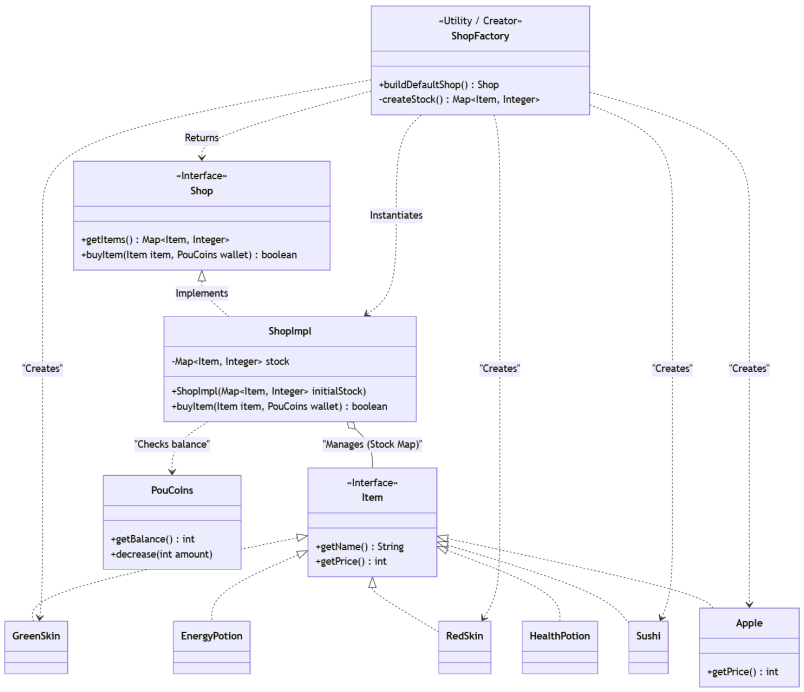

Figura 2.9: Schema UML della creazione estensibile del catalogo tramite ShopFactory

Pattern utilizzati

Simple Factory (idiom) La classe `ShopFactory` incapsula la complessità della creazione degli oggetti del catalogo. Pur non implementando il pattern *Factory Method* dei GoF, che richiederebbe una gerarchia basata sull'ereditarietà, essa agisce come una *Simple Factory* (o factory statica). Questo consente di centralizzare tutte le modifiche al catalogo in un unico punto: ad esempio, per creare un "negozio natalizio" è sufficiente aggiungere un nuovo metodo alla factory, senza modificare il codice di business del negozio o dei controller.

Dependency Injection (via costruttore) Il pattern architetturale dominante in questa sezione è l'*iniezione delle dipendenze* tramite costruttore. Fornendo il catalogo (`Map<Item, Integer>`) al momento dell'istanziazione, si massimizza la coesione della classe `ShopImpl` e si favorisce il suo riuso in contesti differenti, come un negozio specializzato in soli consumabili o uno dedicato esclusivamente alle skin.

Il modello Bedroom e la gestione dinamica dell'aspetto (*Wardrobe*)

Descrizione del problema

Il modello `Bedroom` introduce una problematica di interazione complessa, legata al sottosistema di gestione del *Wardrobe*. Le principali sfide progettuali affrontate sono due.

Filtraggio semantico L'inventario rappresenta un contenitore eterogeneo di oggetti (cibo, pozioni, skin). La `BedroomView` deve visualizzare esclusivamente gli oggetti equipaggiabili (`Skin`), ignorando tutte le altre categorie, senza accoppiarsi alla logica interna o alle implementazioni concrete delle classi `Item`.

Coerenza visiva Il cambio di skin deve riflettersi immediatamente sul componente grafico del personaggio (`PouCharacterView`), mantenendo uno stato visivo coerente senza necessità di ricaricare o ricostruire l'intera scena.

Soluzione proposta

È stato adottato un approccio basato sulla *UI Composition* e sulla navigazione sequenziale dei dati. L'alternativa di generare un bottone per ogni singolo `Item` è stata scartata per evitare di saturare lo spazio visivo e per mantenere l'interfaccia scalabile indipendentemente dalla quantità di oggetti posseduti.

La soluzione scelta prevede che la `BedroomView` riceva una proiezione filtrata dell'inventario. Mediante l'uso delle *Java Stream API* nel controller, la collezione eterogenea viene filtrata funzionalmente (`filter(item -> item instanceof Skin)`) e passata alla vista. La View gestisce internamente una logica a carosello: l'utente scorre la collezione tramite controlli di navigazione statici (*Next*) e conferma la selezione con un comando dedicato (*Wear*). L'azione di equipaggiamento è disaccoppiata tramite un'interfaccia funzionale `Consumer<Skin>`, che delega al `BedroomController` la modifica dello stato nel modello, garantendo una netta separazione delle responsabilità.

Schema UML

Il diagramma UML di riferimento, riportato in Figura 2.10, evidenzia una relazione di composizione: la `BedroomView` contiene un'istanza di `PouCharacterView`, responsabile esclusivamente del rendering grafico del personaggio.

La *View* dipende dall'interfaccia `Inventory` per l'accesso ai dati e dall'interfaccia `InventoryController` per l'invio dei comandi. È rilevante osservare come la `BedroomView` dipenda unicamente dall'interfaccia `Skin`, e non dalle classi concrete (`RedSkin`, `GreenSkin`), garantendo un disaccoppiamento completo in conformità al *Dependency Inversion Principle*.

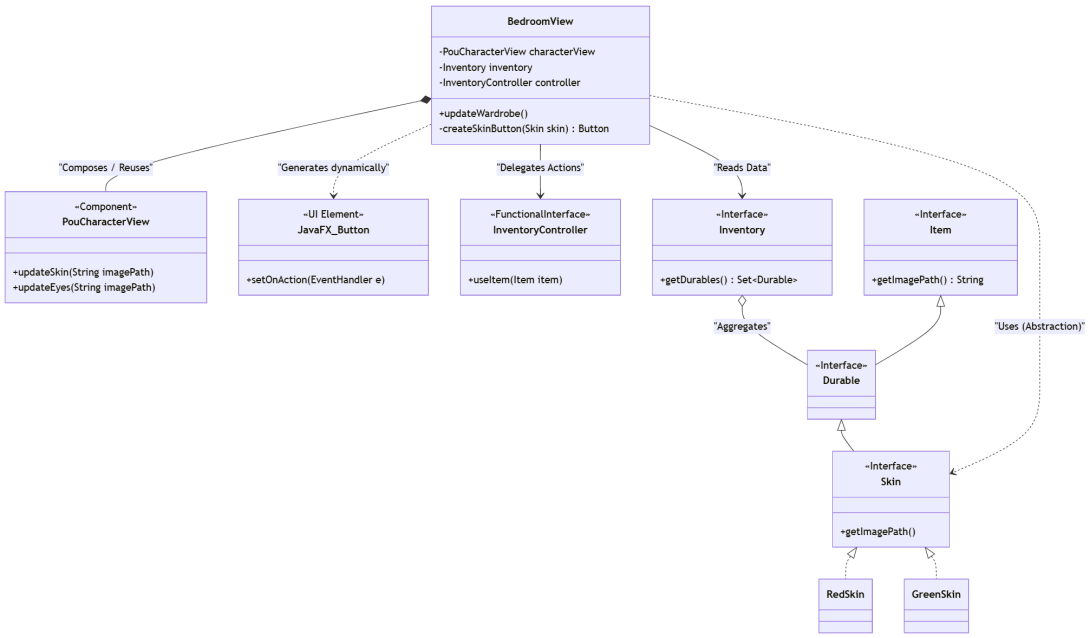

Figura 2.10: Schema UML della gestione dinamica dell'aspetto nel modulo Bedroom

2.2.3 Giovanni Morelli

Architettura dei minigiochi e decomposizione

Problema L'integrazione di un minigioco all'interno dell'applicazione principale presentava una sfida architetturale: il minigioco deve possedere un proprio ciclo di vita (start, update, game over) distinto da quello dell'applicazione principale (il "Tamagotchi"). Inoltre, era necessario garantire l'estendibilità del sistema, permettendo l'aggiunta futura di nuovi minigiochi (es. Snake, Pong) senza dover modificare la logica della `GameRoom` o del `GameRoomController`. Un accoppiamento stretto tra la stanza e una specifica implementazione (es. `FruitCatcherGame`) avrebbe violato il principio *Open/Closed*.

Soluzione proposta Ho adottato il pattern **Strategy**, definendo un'interfaccia comune `Minigame` che astrae il contratto di un qualsiasi gioco arcade integrabile. L'interfaccia espone i metodi fondamentali per il controllo del flusso: `startGame()`, `gameLoop(long now)`, `getScore()`, `isGameOver()` e `calculateCoins()`.

Il `GameRoomController` agisce come *Context* di alto livello, gestendo l'avvio e la chiusura dei giochi in modo polimorfico tramite l'interfaccia. Tuttavia, il controller specifico del minigioco (`FruitCatcherControllerImpl`) mantiene un riferimento alla classe concreta `FruitCatcherGame`. Quando l'utente avvia il minigioco:

- il *Main Game Loop* viene messo in pausa, evitando il decadimento delle statistiche durante la partita;
- viene istanziato e avviato il controller del minigioco;
- al termine della partita (*Game Over*), il controllo ritorna al `MainController`, le monete vinte vengono accreditate al modello `PouLogic` e il loop principale viene riattivato.

```java
public interface Minigame {
    void startGame();
    void gameLoop(long now);
    int getScore();
    boolean isGameOver();
    int calculateCoins();
}
```

Listing 2.9: Interfaccia `Minigame`

Schema UML La Figura 2.11 illustra l'architettura adottata. Si nota che, mentre `FruitCatcherGame` implementa l'interfaccia `Minigame` (rispettando il contratto Strategy), il controller dedicato `FruitCatcherControllerImpl` dipende direttamente dalla classe concreta. Questa scelta pragmatica è stata necessaria per accedere a metodi specifici del gioco (come `getFallingObjects()` per il rendering o `setPlayerPosition()`) che non sono esposti dall'interfaccia generica, garantendo così l'accesso ai dati necessari alla View senza dover ricorrere a cast espliciti o downcasting.

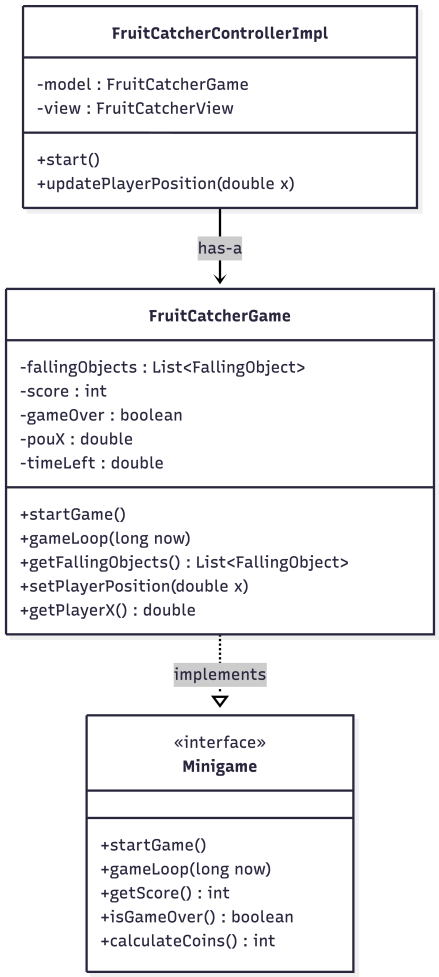

Figura 2.11: Diagramma UML dell'architettura. Notare come il Controller agisca da ponte verso i metodi specifici dell'implementazione concreta, mentre la GameRoom (non mostrata per brevità) gestisce il ciclo di vita tramite l'interfaccia generica.

Alternative considerate Si è valutato l'utilizzo dell'ereditarietà tramite una classe astratta `AbstractMinigame` (Pattern *Template Method*) invece dell'interfaccia pura. Sebbene avrebbe permesso di condividere codice comune (es. gestione del punteggio base), si è preferita l'interfaccia per favorire la composizione e non vincolare i futuri minigiochi a una struttura rigida, dato che giochi diversi (es. un gioco di carte rispetto a un platform) potrebbero non condividere alcuna logica di implementazione.

Game loop e gestione fisica

Problema Il minigioco *Fruit Catcher* richiede un aggiornamento costante dello stato di decine di entità (frutta e bombe), indipendentemente dalla frequenza di rendering. Era necessario separare l'aggiornamento logico dalla visualizzazione in modo da mantenere il gameplay fluido a 60 FPS e deterministico.

Soluzione proposta La logica del gioco è incapsulata nella classe `FruitCatcherGame`. Il cuore del sistema è il metodo `gameLoop(long now)`, invocato periodicamente tramite un `AnimationTimer`. All'interno di questo loop avviene l'aggiornamento dello stato:

1. **Aggiornamento timer**: il tempo di gioco residuo viene decrementato a ogni frame.
2. **Spawning**: nuovi oggetti (`FallingObject`) vengono generati casualmente sulla base di una probabilità configurabile (`SPAWN_PROBABILITY`).
3. **Fisica**: ogni oggetto aggiorna la propria posizione verticale applicando una costante di gravità.
4. **Collisioni**: il metodo privato `updateAndCheckCollisions()` verifica l'intersezione tra la bounding box del giocatore e quella degli oggetti cadenti.

La classe `FallingObject` delega la definizione delle proprietà di gioco (punteggio e pericolosità) al suo enum interno `Type`, che definisce quattro varianti: `APPLE`, `PINEAPPLE`, `BANANA` e `BOMB`.

```
@Override
public void gameLoop(final long now) {
    if (this.gameOver) {
        return;
    }

    // Timer logic
    this.timeLeft -= TIME_DECREMENT;
    if (this.timeLeft <= 0) {
        this.gameOver = true;
        return;
    }

    // Spawn objects
    if (this.random.nextDouble() < SPAWN_PROBABILITY) {
            spawnObject();
        }

    // Collisions
    updateAndCheckCollisions();
}
```

Listing 2.10: Logica del game loop in `FruitCatcherGame`

Schema UML La Figura 2.12 mostra la struttura delle classi coinvolte nella simulazione fisica. `FruitCatcherGame` agisce da gestore del ciclo di vita per la lista di `FallingObject`, i quali incapsulano stato fisico (posizione) e semantico (tipo).

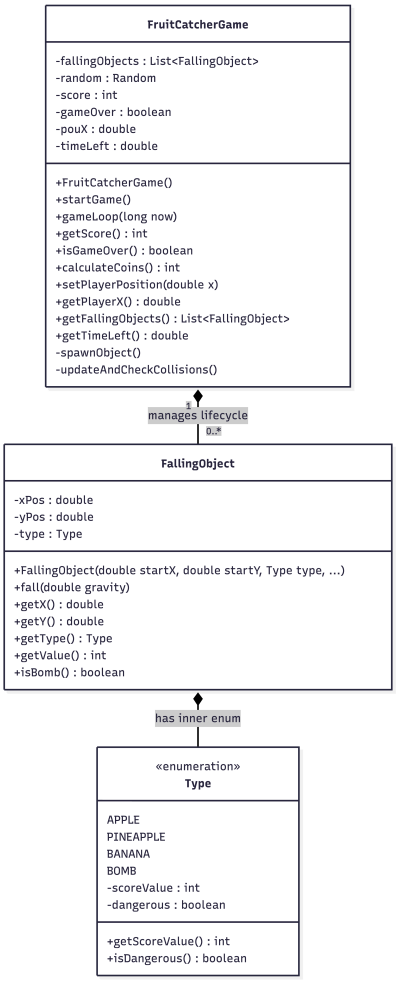

Figura 2.12: Diagramma delle classi per la gestione fisica. FruitCatcherGame gestisce le istanze di FallingObject e il loro enum interno Type. Mentre il diagramma precedente (Fig. 2.11) illustrava l'architettura ad alto livello e le interazioni esterne della classe `FruitCatcherGame`, il seguente diagramma (Fig. 2.12) entra nel dettaglio implementativo della stessa, esplodendo le relazioni con le entità di gioco gestite internamente

Crescita reattiva del personaggio (Reactive Observer)

Problema Una delle feature principali del gioco è l'invecchiamento del Pou, che deve riflettersi visivamente nelle sue dimensioni. Gestire manualmente l'aggiornamento della vista ogni volta che l'età cambia (chiamando un metodo `updateSize()`) avrebbe introdotto codice ridondante e accoppiato logicamente il controllore del tempo alla vista.

Soluzione proposta Ho sfruttato il sistema di *Binding* e *Properties* di JavaFX per implementare un pattern **Observer** in forma reattiva. Nella classe `PouCharacterView`, il metodo `bindSize` accetta la `IntegerProperty` dell'età esposta direttamente dalla classe `PouLogic` (il Model). Viene registrato un `ChangeListener` che ricalcola il fattore di scala del nodo grafico (`scaleX`, `scaleY`) ogni volta che il valore dell'età varia.

La formula di interpolazione lineare mappa l'età corrente (da 0 a `MAX_AGE_REF`) in un fattore di scala compreso tra `MIN_SCALE` (0.5) e `MAX_SCALE` (2.0).

```
public void bindSize(final IntegerProperty ageProperty) {
    if (ageProperty != null) {
        ageProperty.addListener((_, _, newAge) ->
            updateScale(newAge.intValue())
        );
        updateScale(ageProperty.get()); // Initial update
    }
}
```

Listing 2.11: Binding della crescita in `PouCharacterView`

Questo meccanismo garantisce che la rappresentazione visiva del personaggio sia sempre sincronizzata con lo stato logico del Model, senza che il `MainController` debba orchestrare esplicitamente l'aggiornamento.

Schema UML La Figura 2.13 mostra il legame reattivo: la vista si lega alla proprietà esposta dal modello `PouLogic`. Si noti come l'età sia gestita come una proprietà semplice e non come una `PouStatistic`, poiché non richiede limiti superiori rigidi o meccanismi di clamping complessi.

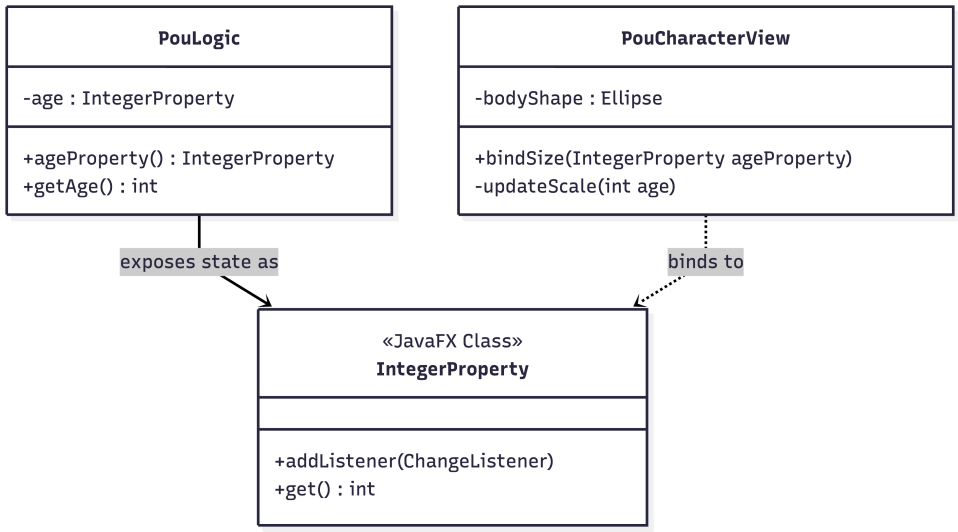

Figura 2.13: Diagramma del pattern Observer. PouCharacterView osserva la proprietà dell'età esposta da PouLogic tramite binding unidirezionale.

Alternative considerate L'alternativa era un approccio **Imperativo**: il `GameLoop` avrebbe dovuto invocare esplicitamente `view.setSize(newSize)` ad ogni scatto dell'età. Questo avrebbe violato il principio di *Separation of Concerns*, costringendo la logica di dominio a preoccuparsi di dettagli di presentazione. L'approccio reattivo (Binding) inverte la dipendenza: è la vista a reagire ai dati, rendendo il sistema più robusto.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il progetto include una suite di test automatizzati con **JUnit 5**, focalizzati sulla logica del modello e sul comportamento del Pou. I test verificano che le funzionalità principali funzionino come previsto e aiutino ad evitare regressioni quando si modificano le classi.

Componenti testate

- `PouLogic`: test delle azioni del Pou (mangiare, dormire, svegliarsi, lavarsi, giocare) e dei vincoli di stato, come l'impossibilità di mangiare quando dorme o di compiere azioni quando è morto.
- `PouState` e `PouStatistics`: verifica delle transizioni di stato (AWAKE, SLEEPING, DEAD) in base alle statistiche e del rispetto dei bounds per Fame, Energia, Salute e Divertimento (pattern clamping 0-100).
- `PouStatisticsDecay`: test della logica pura del decadimento temporale tramite il metodo `performDecay()`. Il metodo è testato passando istanze concrete delle statistiche, permettendo di simulare l'invecchiamento e il cambio di stato senza attendere il tempo reale del game loop.
- `Inventory` e `Shop`: controllo di aggiunta/rimozione oggetti nell'inventario, acquisti nello shop, verifica fondi insufficienti e gestione della proprietà di oggetti unici (skin).
- **Minigiochi**: test della logica del modello `FruitCatcherGame` (movimento, confini, calcolo punteggio e Game Over), indipendentemente dall'interfaccia grafica.
- **Persistenza**: test di `PersistenceManager` per la serializzazione/deserializzazione JSON e la corretta ricostruzione dello stato del modello dai record `PouSaveData`.

Strategie di testing Per test rapidi e deterministici, si evita `Thread.sleep()`: in `PouStatisticsDecayManager` il metodo `performDecayTick()` simula istantaneamente un ciclo temporale, permettendo di verificare il decadimento senza aspettare secondi reali.

I test usano il pattern *Arrange–Act–Assert*: si prepara il `Pou` con statistiche note, si chiama il metodo da testare (es. `eat()` e si controllano i risultati con `Assertions`.

Code Coverage con JaCoCo È integrato il plugin **JaCoCo** in `build.gradle.kts`, che genera report HTML dopo `./gradlew test` nella cartella `build/jacocoHtml`.

I report mostrano:

- Percentuale di righe coperte dai test.
- Copertura dei rami condizionali.
- Parti del codice non testate, per aggiungere casi limite.

3.2 Note di sviluppo

3.2.1 Diego Andruccioli

Elenco delle feature avanzate del linguaggio e dell'ecosistema Java utilizzate nella parte del progetto da me sviluppata:

- **Record Java**: DTO immutabili per la persistenza dei dati (`PouSaveData`, `SavedStatistics`, `SavedInventory`, `SavedItem`). Utilizzo di costruttori compatti per la programmazione difensiva (copie di liste mutabili).
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/model/save/PouSaveData.java
- **Pattern Matching for Switch**: Espressioni switch per la mappatura concisa tra Enum e implementazioni, utilizzata in `MainControllerImpl` (mappatura RoomView) e `PersistenceControllerImpl` (creazione Item da stringa).
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/controller/PersistenceControllerImpl.java
- **Unnamed Variables (`_`)**: Utilizzo della sintassi per ignorare parametri non necessari nelle lambda expression, migliorando la leggibilità (es. in `MainControllerImpl` e listener JavaFX).
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/controller/MainControllerImpl.java
- **Enum types**: `PouState` e `Room` utilizzati per la gestione degli stati vitali e della navigazione, garantendo type-safety e centralizzazione delle costanti.
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/model/PouState.java
- **Lambda expressions**: Ampio utilizzo per la gestione eventi JavaFX (es. `Button.setOnAction`) e per i listener del game loop in `GameLoop`.
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/controller/room/GameRoomController.java
- **Method references**: Riferimenti a metodi per codice più pulito come `this::tick` nel `ScheduledExecutorService` e `this::updateGlobalStatistics` in `Platform.runLater`.
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/controller/PouGameLoop.java

- **Functional interfaces**: Utilizzo di interfacce standard come `Runnable` (game loop), `Consumer` (cambio skin in `BedroomView`) e `Supplier` (lazy initialization dei controller).
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/controller/MainControllerImpl.java

- **JavaFX Properties e Bindings**: Utilizzo di `IntegerProperty` e `ObjectProperty` per il binding reattivo tra Model e View (es. aggiornamento età e stato sonno/veglia).
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/model/PouLogic.java

- **ScheduledExecutorService**: Gestione del game loop e del tempo di gioco tramite `Executors.newSingleThreadScheduledExecutor()`, con gestione della concorrenza tramite `CopyOnWriteArrayList`.
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/controller/PouGameLoop.java

- **GSON**: Serializzazione JSON per la persistenza dei dati, gestita tramite libreria esterna con configurazione `GsonBuilder`.
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/persistence/PersistenceManager.java

- **Java Logging API (JUL)**: Sistema di logging nativo (`java.util.logging`) per il tracciamento delle operazioni critiche e debug.
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/src/main/java/it/unibo/jpou/mvc/controller/MainControllerImpl.java

- **JUnit 5**: Framework di testing utilizzato per verificare la logica di dominio e i componenti modello.
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/build.gradle.kts

- **JaCoCo**: Strumento per l'analisi della code coverage integrato tramite Gradle.
  Permalink: https://github.com/diegoandruccioli/PSS25-JPOU/blob/main/build.gradle.kts

Note su codice riadattato e utilizzo AI Per lo sviluppo di questo progetto ho recuperato e riadattato parte del codice proveniente dal repository https://github.com/diegoandruccioli/ERROR-JPOU, frutto di un precedente tentativo di sviluppo da parte del medesimo gruppo di lavoro, successivamente sospeso per criticità strutturali (si veda il paragrafo 4.2 per i dettagli). L'architettura attuale è stata quindi rifondata partendo dai modelli di riferimento forniti dai docenti. L'intelligenza artificiale è stata impiegata come strumento di supporto in due modalità principali:

- **Generazione mirata di codice**: Nel progetto iniziale, ho utilizzato l'AI per abbozzare la logica di tracciamento oculare (`setUpEyeTracking` e `updatePupil`) all'interno delle view: tale logica è stata successivamente integrata e adattata manualmente anche nell'attuale ricostruzione del progetto.

Nel progetto attuale è stata utilizzata per implementare la logica di **pausa precisa** all'interno del `PouGameLoop`, permettendo al sistema di tenere traccia del tempo trascorso durante i minigiochi e riprendere il decadimento delle statistiche esattamente dal punto di interruzione, evitando reset o imprecisioni temporali.

- **Tutor per spiegazione e sviluppo**: L'AI ha svolto un ruolo fondamentale come tutor interattivo per chiarire dubbi teorici e pratici durante lo sviluppo:
  - Ha fornito spiegazioni dettagliate sulla gestione della thread-safety in JavaFX, in particolare sulla corretta gestione della concorrenza tra `Platform.runLater` e `ScheduledExecutorService`;
  - Ha chiarito l'utilizzo corretto dei Record Java per garantire l'immutabilità dei dati;
  - Ha suggerito strategie di refactoring per migliorare la validazione dei dati all'interno del sistema di persistenza;
  - Ha fornito pareri sulla fattibilità tecnica di specifiche scelte implementative, valutando alternative più idonee al contesto del progetto, che ho poi implementato autonomamente basandomi sui consigli ricevuti (senza ricorrere a codice generato).

3.2.2 Rei Mici

Di seguito è riportato l'elenco delle feature avanzate del linguaggio Java e del relativo ecosistema utilizzate nella mia parte di progetto.

- **Java Stream API**: È stato fatto uso estensivo delle Stream API per la manipolazione delle collezioni, in particolare per operazioni di filtraggio semantico e mapping (proiezione dei dati) senza effetti collaterali. Questo costrutto funzionale sostituisce iterazioni imperative complesse, migliorando la leggibilità e riducendo il rischio di errori logici nei cicli.
  Riferimento: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/controller/room/BedroomController.java#L86-L89

- **Interfacce Funzionali del JDK (`java.util.function`)**: Utilizzo delle interfacce generiche `Supplier<T>` e `Consumer<T>`. In particolare, il `Supplier` è stato impiegato per implementare il caricamento *lazy* delle dipendenze nel Controller, evitando l'inizializzazione prematura di componenti pesanti.
  Riferimento: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/controller/inventory/InventoryControllerImpl.java#L19-L20

- **Lambda Expressions e Method References**: Utilizzo di espressioni lambda e riferimenti a metodo (es. `this::buyItem`) per definire in modo conciso i gestori di eventi (Event Handlers) e le callback della UI, trattando il codice come dato ed evitando la verbosità delle classi anonime interne.
  Riferimento: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/view/room/ShopView.java#L69

- **Internal Iteration (`Map.forEach`)**: Adozione dei metodi di iterazione interna introdotti in Java 8 per le interfacce `Map`, che permettono di applicare una logica `BiConsumer` direttamente sulle entry della mappa, eliminando l'esplicita gestione degli `EntrySet`.
  Riferimento: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/view/room/ShopView.java#L64

- **Libreria Grafica Esterna (JavaFX) e Integrazione CSS**: Utilizzo dell'ecosistema JavaFX per la costruzione dello *Scene Graph*, con impiego di controlli complessi e gestione del layout tramite `BorderPane` e `FlowPane`. Lo styling è delegato a file CSS esterni caricati dinamicamente a runtime tramite `ClassLoader`.
  Riferimento: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/view/room/AbstractRoomView.java#L33

Provenienza del Codice e Integrazione dell'Intelligenza Artificiale

Recupero e Refactoring Architetturale

Il progetto ha subito un parziale riadattamento strutturale a partire dal repository `ERROR-JPOU`. A seguito della sospensione del precedente tentativo per criticità sistemiche, l'architettura attuale è stata rifondata *ex novo*. La logica di dominio è stata riallineata ai pattern architetturali MVC e ai requisiti di scalabilità e manutenibilità definiti dai docenti.

Supporto Metodologico dell'Intelligenza Artificiale

L'Intelligenza Artificiale è stata impiegata come strumento di supporto tecnico e teorico, garantendo la piena originalità dell'opera attraverso i seguenti contributi:

Modernizzazione e Code Safety Riduzione del debito tecnico mediante attività di refactoring sintattico e la correzione di violazioni rilevate da SpotBugs. In particolare, si segnala l'adozione di `Supplier<T>` nei Controller (ad esempio in `ShopControllerImpl`) per neutralizzare l'esposizione della rappresentazione interna (EI2), nonché la conversione di logiche imperative in pipeline funzionali basate sulla *Stream API*, con l'obiettivo di migliorare la leggibilità e la manutenibilità dei moduli *Inventory* e *Shop*.

Validazione Architetturale Utilizzo dell'AI come revisore critico per la verifica dell'implementazione delle strategie di difesa dell'immutabilità (tramite *Unmodifiable Collections* nei getter di `InventoryImpl` e `ShopImpl`) e per il corretto disaccoppiamento dei livelli MVC.

3.2.3 Giovanni Morelli

Elenco delle feature avanzate del linguaggio e dell'ecosistema Java utilizzate nella mia parte del progetto:

- **JavaFX AnimationTimer e Inner Classes**: Per la gestione del *Game Loop*, ho evitato l'uso di thread standard gestiti manualmente, preferendo la classe astratta `AnimationTimer` di JavaFX. Ho implementato il loop come una *Inner Class* (`InternalLoop`) all'interno del controller per incapsulare la logica di aggiornamento frame-by-frame, mantenendo accesso ai campi privati della classe esterna ma

nascondendo il dettaglio implementativo del loop al resto dell'applicazione.
Permalink: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/controller/minigames/FruitCatcherControllerImpl.java

- **JavaFX Canvas API (Immediate Mode Rendering)**: Per il rendering degli oggetti ad alta frequenza di aggiornamento (frutta e bombe), ho utilizzato l'approccio *Immediate Mode* tramite `Canvas` e `GraphicsContext`. Questo ha richiesto l'uso avanzato dello stack grafico (`save()` e `restore()`) per isolare le trasformazioni geometriche (rotazioni e traslazioni) di ogni singolo sprite, evitando l'overhead prestazionale del *Retained Mode* (Scene Graph) che si avrebbe istanziando nodi `ImageView` multipli.
Permalink: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/view/minigames/FruitCatcherJavaFXView.java

- **Functional Interfaces e Callbacks (`Consumer`)**: Ho sfruttato le interfacce funzionali (es. `Consumer<Integer>`) per implementare meccanismi di *callback* tra il controller del minigioco e il controller principale, disaccoppiando la logica di assegnazione delle monete. Inoltre, ho fatto ampio uso di espressioni lambda per definire in modo conciso gli *event handler* (es. input tastiera) e i listener reattivi.
Permalink: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/controller/minigames/FruitCatcherControllerImpl.java

- **Observer Pattern (via JavaFX Properties)**: Ho applicato il pattern Observer sfruttando le `IntegerProperty` di JavaFX (in particolare per il campo `age` in `PouLogic`). Tramite il meccanismo di binding unidirezionale, ho sincronizzato la dimensione della vista del Pou con il valore logico dell'età, disaccoppiando completamente la logica di invecchiamento dalla sua rappresentazione grafica reattiva.
Permalink: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/model/PouLogic.java

- **Enum Types con Stato e Comportamento**: L'enumerazione `FallingObject.Type` non è stata usata come semplice lista di costanti, ma come classe contenente stato immutabile (punteggio, flag di pericolosità) e metodi accessori. Questo ha permesso di centralizzare la configurazione delle entità di gioco, sfruttando il polimorfismo dei dati ed evitando catene di `switch-case` ridondanti nel codice client.
Permalink: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/model/minigames/fruitcatcher/FallingObject.java

- **Gestione sicura delle Collezioni (Iterator)**: Nel ciclo di aggiornamento fisico, ho utilizzato esplicitamente un `Iterator` per scorrere la lista degli oggetti cadenti. Sebbene l'uso di costrutti più moderni come `Collection.removeIf()` o gli Stream avrebbe reso il codice più sintetico, l'iteratore esplicito è stato preferito per ragioni di efficienza: esso consente di eseguire l'aggiornamento della fisica, il rilevamento delle collisioni e la rimozione condizionale (`iterator.remove()`) in un unico passaggio (*single pass*), evitando di dover iterare sulla lista più volte per frame e prevenendo la `ConcurrentModificationException`.
Permalink: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/

main/java/it/unibo/jpou/mvc/model/minigames/fruitcatcher/FruitCatcherGame.java

- **Dependency Injection e Annotazioni**: Ho applicato il principio di *Dependency Injection* tramite costruttore nel `FruitCatcherControllerImpl`, ricevendo le dipendenze esterne (View e Callback) al momento dell'istanziazione per favorire la testabilità. Inoltre, ho fatto uso di annotazioni per l'analisi statica come `@SuppressFBWarnings` per documentare e gestire consapevolmente i casi limite rilevati da SpotBugs.
  Permalink: https://github.com/diegoandruccioli/pss25-jpou/blob/main/src/main/java/it/unibo/jpou/mvc/controller/minigames/FruitCatcherControllerImpl.java

Note su codice riadattato e utilizzo AI Non ho riadattato codice proveniente da progetti esterni o librerie di terze parti; l'architettura del minigioco e la gestione della View su Canvas sono state scritte da zero. L'utilizzo di strumenti di AI generativa è stato limitato a due ambiti specifici:

- **Generazione di snippet algoritmici**: L'AI è stata utilizzata come supporto per implementare le parti matematicamente più complesse, in particolare la gestione corretta dello stack grafico (`save/restore`) nel `GraphicsContext` e la logica fisica di accelerazione (gravità) all'interno del loop, che richiedevano una precisione specifica nel calcolo delta-time. Il codice generato è stato poi integrato manualmente nella struttura MVC del progetto.
- **Ruolo di tutor tecnico**: Ho interrogato l'AI per comprendere a fondo le differenze prestazionali e architetturali tra *Scene Graph* e *Canvas* in JavaFX, e per risolvere problemi complessi legati alla sincronizzazione tra il Timer del gioco e l'aggiornamento della UI (come il problema delle "doppie monete" generato da race condition sugli eventi).

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Diego Andruccioli

Ho sviluppato l'intero ciclo vitale dell'entità Pou, gestendo le statistiche vitali e il sistema economico (`PouStatistics`, `PouCoins`), la logica di decadimento temporale (`PouGameLoop` e `PouStatisticsDecay`) e la gestione centrale degli stati (`PouState`). Mi sono occupato inoltre dell'architettura delle stanze "Camera da letto" e "Bagno" (con relativi Controller e View) e dell'intero sottosistema di persistenza (`PersistenceManager` e i record DTO).

Il flusso di sviluppo ha seguito un approccio incrementale: sono partito dalla modellazione delle statistiche (Fame, Energia, Salute, Divertimento). Successivamente ho implementato il motore di decadimento per garantire il realismo delle meccaniche di gioco e ho integrato le stanze di mia competenza nella View principale per validare le interazioni. Infine, ho realizzato il sistema di persistenza JSON per garantire il salvataggio completo dello stato (statistiche, inventario, posizione).

Un momento critico dello sviluppo è stato l'abbandono di un template iniziale rivelatosi inadeguato, che ci ha costretti a riscrivere il progetto da zero. In questa fase ho assunto un ruolo chiave nelle decisioni architetturali (struttura del Model, serializzazione GSON), lavorando in stretta sinergia con il resto del team. L'ultima settimana è stata caratterizzata da un intenso lavoro di gruppo per rispettare la scadenza, durante il quale il plugin di analisi statica fornito dal docente è stato fondamentale per guidarci verso una corretta implementazione e risolvere tempestivamente le criticità tecniche. Questo imprevisto ha trasformato il progetto in una vera prova di resilienza e collaborazione.

In fase di autovalutazione, considero un punto di forza il netto disaccoppiamento ottenuto tra Logica e Tempo e tra Dominio e Persistenza. Una criticità iniziale, già riscontrata nel precedente tentativo, è stata la gestione della complessità nel `MainController`, risolta efficacemente applicando il principio di delega ai sotto-controller specifici (es. `BathroomController`, `OverlayController`).

Per sviluppi futuri, il progetto trarrebbe beneficio dall'introduzione di un sistema di animazioni per rendere l'interazione con il Pou più dinamica e coinvolgente.

4.1.2 Rei Mici

Il mio contributo all'interno del progetto si è focalizzato sulla progettazione e implementazione del *Core Gestionale*, inteso come l'infrastruttura logica responsabile della gestione

dell'economia di gioco, delle risorse e della persistenza degli oggetti. In particolare, ho assunto la responsabilità dell'intero ciclo di vita delle entità, dalla loro definizione gerarchica e creazione tramite il modello `Shop`, fino alla gestione all'interno dell'`Inventory` e al successivo utilizzo contestuale nelle diverse stanze, quali `Kitchen` e `Infirmary`. Inoltre, ho seguito in modo verticale lo sviluppo del sottosistema della `Bedroom`, affrontando le problematiche specifiche legate alla gestione dinamica del guardaroba e alla coerenza visiva del personaggio.

Ripercorrendo le scelte progettuali adottate, il principale punto di forza del lavoro risiede nella robustezza architetturale del modello dati. La decisione di dividere strutturalmente gli oggetti in `Consumable` e `Durable`, supportata dall'impiego di strutture dati dedicate (`Map` e `Set`) che impongono autonomamente le regole di accumulo e unicità, ha garantito un elevato livello di *type safety*. Tale impostazione ha consentito di eliminare alla radice intere classi di bug logici e di incoerenza dei dati. Parallelamente, l'adozione di pattern quali la *Factory* per il popolamento del negozio e l'approccio *data-driven* per la generazione dell'interfaccia del guardaroba ha conferito al sistema un'elevata estendibilità, nel rispetto del *Principio Open/Closed*: l'architettura risulta predisposta all'introduzione di nuovi contenuti senza richiedere modifiche strutturali al codice esistente.

Un'analisi critica del prodotto evidenzia tuttavia alcuni margini di miglioramento. In particolare, l'attuale gestione della persistenza, basata sulla serializzazione completa dello stato applicativo in formato testuale/JSON, sebbene adeguata per la mole di dati attuale, potrebbe rivelarsi inefficiente in scenari caratterizzati da carichi elevati o da inventari di grandi dimensioni. In tali contesti, un approccio fondato su salvataggi incrementali o sull'integrazione di un database locale (ad esempio `SQLite`) avrebbe potuto offrire migliori garanzie in termini di prestazioni e scalabilità. Inoltre, la gerarchia degli oggetti basata su ereditarietà classica, pur risultando solida, presenta una certa rigidità strutturale: l'eventuale introduzione di oggetti *ibridi* (ad esempio equipaggiamenti a consumo temporale) richiederebbe un refactoring significativo delle interfacce di base.

In prospettiva futura, qualora il progetto evolvesse verso un prodotto completo, la logica di sviluppo si concentrerebbe principalmente sull'espansione del sistema di *Housing* e dell'economia di gioco. Risulterebbe prioritario estendere la logica dei `Durable` per supportare non solo le skin del personaggio, ma anche oggetti di arredamento liberamente posizionabili all'interno delle stanze, trasformando le viste statiche in veri e propri editor interattivi. Sul piano economico, l'introduzione di meccaniche di rivendita (*Sell*) e di fluttuazioni dinamiche dei prezzi contribuirebbe ad aumentare la profondità del gameplay. Infine, dal punto di vista della logica di gioco, l'evoluzione naturale del sistema prevederebbe l'introduzione di effetti visivi nel momento dell'utilizzo di consumabili.

4.1.3 Giovanni Morelli

Il mio contributo principale al progetto ha riguardato la progettazione e l'implementazione del sottosistema dei minigiochi con la relativa scalabilità, la gestione della logica evolutiva del personaggio (crescita) e la realizzazione della vista "Sala Giochi".

In fase di autovalutazione, identifico come principale **punto di forza** la gestione tecnica del rendering e del loop di gioco. La sfida maggiore è stata orchestrarw l'interazione tra componenti eterogenei, facendo convivere un *Game Loop* attivo (tipico dei videogiochi) con il ciclo di vita reattivo dell'interfaccia JavaFX. Ho dovuto approfondire la gestione della concorrenza per sincronizzare il thread di gioco con l'`Application Thread`, optando per l'implementazione della vista `FruitCatcherJavaFXView` tramite API `Canvas`

(rendering in *Immediate Mode*). Questa scelta, preferita allo *Scene Graph* standard, ha permesso di ottenere prestazioni stabili a 60 FPS, evitando l'overhead legato alla gestione di numerosi nodi `ImageView` per oggetti a vita breve.

Un momento decisivo per la qualità del mio lavoro è coinciso con la fase di ristrutturazione totale del progetto. Il dover re-implementare la logica in tempi ristretti, dopo aver abbandonato l'architettura iniziale, mi ha imposto un rigore assoluto nel rispetto della separazione MVC. Questo "refactor forzato" è stato l'esercizio più formativo, trasformando una difficoltà critica in un'architettura finale solida e ben disaccoppiata.

Per quanto riguarda il mio ruolo nel gruppo, durante lo sviluppo autonomo delle componenti, il confronto continuo con i colleghi è risultato fondamentale. In particolare, prima di effettuare operazioni di merge tra branch Git, ci siamo spesso coordinati tramite incontri dal vivo o collegamenti online per evitare conflitti o perdite di codice. Sebbene l'uso di Git abbia inizialmente rallentato il processo essendo uno strumento nuovo per noi, si è rivelato poi essenziale per la gestione del lavoro collaborativo.

Un **punto di debolezza** risiede nella semplicità attuale delle meccaniche del minigioco: sebbene l'architettura sia solida (grazie allo *Strategy Pattern*), il gameplay risulta basilare e privo di una progressione di difficoltà (livelli o velocità incrementale).

In un'ottica di **sviluppi futuri**, il progetto si presta naturalmente all'espansione della "Sala Giochi". Grazie all'interfaccia `Minigame` già implementata, sarebbe immediato aggiungere nuovi titoli arcade (es. Snake, Pong) e introdurre un sistema di classifiche locali o online per aumentare la longevità dell'applicazione.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Diego Andruccioli

Il percorso di sviluppo è stato caratterizzato da sfide significative e da un'importante crescita personale. Inizialmente, mi sono scontrato con la difficoltà di tradurre i concetti teorici in un'architettura concreta: pur avendo chiari i singoli pattern, l'orchestrazione di un sistema da zero ha generato inizialmente molti dubbi sulla correttezza delle scelte implementative. Il primo vero punto di svolta è stato vedere i miei componenti integrarsi con successo con quelli dei compagni durante le prime fasi di merge.

La criticità maggiore è emersa quando ci siamo resi conto che l'architettura su cui avevamo basato l'intero lavoro (derivata da un template iniziale rivelatosi non idoneo, vedi repository https://github.com/diegoandruccioli/ERROR-JPOU) presentava difetti strutturali insanabili. Questo momento di crisi, anziché scoraggiarci, ha innescato una reazione costruttiva. Il gruppo ha deciso di riscrivere il progetto ex novo, correggendo gli errori alla radice.

In questa fase di sviluppo il supporto la Wiki e l'AI (in veste di tutor tecnico come dettagliato nel paragrafo 3.2) si sono rivelati utili per risolvere dubbi puntuali e comprendere rapidamente le segnalazioni degli strumenti di analisi.

In conclusione, l'esperienza, assai provante, è stata altamente formativa proprio grazie alla gestione degli imprevisti. Come suggerimento per le future edizioni, ritengo sarebbe estremamente utile introdurre laboratori pratici guidati su progetti "full-stack" di medie dimensioni (Model + View + Controller + Persistenza), per colmare il divario tra la teoria dei singoli componenti e la loro integrazione. Inoltre, consiglio vivamente di introdurre l'uso del plugin di validazione (del prof. Pianini) fin dai primi laboratori: abituarsi subito

a standard qualitativi elevati previene l'accumulo di debito tecnico e forma una mentalità di sviluppo rigorosa fin dal primo giorno.

4.2.2 Rei Mici

Sottoscrivo pienamente quanto espresso dai colleghi in merito alla criticità del *refactoring radicale* affrontato a ridosso della consegna finale. Ritengo tuttavia opportuno aggiungere una riflessione personale sul percorso complessivo svolto.

Nonostante possedessi un background tecnico pregresso, maturato durante gli studi superiori e tale da garantirmi una discreta familiarità con i paradigmi della programmazione a oggetti, l'impatto con questo progetto si è rivelato inaspettatamente complesso sotto il profilo metodologico. L'esperienza mi ha permesso di comprendere come lo sviluppo software in un contesto strutturato trascenda la mera implementazione tecnica: la difficoltà principale non è risieduta nella scrittura del codice in sé, bensì nel rispetto rigoroso dei vincoli architetturali, delle scadenze temporali e nella gestione delle dinamiche di lavoro in team.

Nel mio contributo, occupandomi prevalentemente del *Core Gestionale* (Inventario, Shop ed Economia), ho riscontrato una criticità significativa nella fase di transizione tra il *Design* e l'*Implementazione*. Una volta completato il diagramma UML iniziale, ho sperimentato un marcato senso di disorientamento: pur avendo chiara la struttura astratta del sistema, ho incontrato difficoltà nel tradurre le relazioni UML in un'architettura concreta di file e package coerente all'interno dell'IDE.

Superato questo iniziale blocco, la sfida tecnica si è spostata sul mantenimento del rigore del pattern *Model–View–Controller (MVC)* in un contesto caratterizzato da dati eterogenei. È stato necessario contrastare l'istinto di adottare un approccio procedurale orientato al "funzionare subito", per abbracciare invece un design basato su interfacce e disaccoppiamento, come nel caso dell'utilizzo di un `InventoryController` funzionale. Questa disciplina progettuale, sebbene inizialmente frustrante, si è rivelata determinante durante la fase di riscrittura: disporre di una logica di dominio (*Model*) isolata e testabile ha consentito di modificare l'interfaccia grafica senza compromettere le regole fondamentali del gioco.

Un supporto fondamentale nell'affrontare tali complessità è derivato dall'utilizzo dell'Intelligenza Artificiale. L'IA è stata impiegata non come generatore passivo di codice basato sul semplice "copia–incolla", bensì come un vero e proprio *tutor virtuale*. Attraverso interrogazioni mirate, è stato possibile chiarire dubbi concettuali, comprendere errori di compilazione poco espliciti e affrontare attività di debugging avanzato. Questo approccio ha consentito di superare situazioni di stallo in autonomia, trasformando le difficoltà tecniche in opportunità di apprendimento mirato e colmando in tempo reale lacune specifiche sul linguaggio Java.

Parallelamente, l'impatto con gli strumenti di *Qualità del Codice* (QA, Checkstyle, PMD) è risultato inizialmente traumatico ma, a posteriori, fortemente formativo. Sebbene inizialmente percepiti come vincoli burocratici, tali strumenti si sono rivelati essenziali per mantenere il codice manutenibile nel medio periodo; in loro assenza, la prima versione del progetto sarebbe rapidamente degenerata, compromettendo il successo del refactoring finale.

Come suggerimento per le future edizioni del corso, mi associo alla richiesta di laboratori maggiormente *integrati*. In particolare, risulterebbe estremamente utile prevedere una sessione dedicata al *project setup*, che accompagni il passaggio dal diagramma UML

allo scheletro del progetto nell'IDE, nonché esercitazioni specifiche sull'*orchestrazione dei pattern architetturali*. Comprendere come far interagire correttamente componenti diversi, ad esempio collegando una *Factory* a una *Strategy* mantenendo pulita l'architettura, permetterebbe di colmare efficacemente il divario tra la teoria dei singoli pattern e la realizzazione del progetto finale.

4.2.3 Giovanni Morelli

Condivido l'analisi fatta dal collega Diego riguardo al percorso accidentato che ha caratterizzato questo progetto. Il momento di rottura, in cui abbiamo deciso di abbandonare l'architettura iniziale colma di errori strutturali per riscrivere l'applicazione da zero, è stato senza dubbio la sfida più grande. Per la mia parte, questo ha significato dover re-implementare la logica dei minigiochi e del rendering in tempi ristrettissimi, assicurandomi però questa volta di rispettare rigorosamente la separazione MVC che era mancata nel primo tentativo. Sebbene fonte di notevole stress, dato il carico cognitivo di una sessione d'esami particolarmente densa, questo "refactor forzato" si è rivelato paradossalmente l'esercizio più formativo, obbligandoci a capire davvero il *perché* di certe scelte architettoniche.

Per quanto riguarda il corso e gli strumenti adottati, l'impatto iniziale con la scrittura da zero dei primi punti della relazione e schemi UML è stato spiazzante. L'adozione di standard qualitativi molto stringenti (come quelli imposti dai plugin di validazione statici) hanno rappresentato una curva di apprendimento ripida. Inizialmente, la rigorosa attenzione agli aspetti formali richiesta dalle linee guida è sembrata sottrarre tempo allo sviluppo della logica; tuttavia, riconosco che tale rigore ci ha fornito una "rete di sicurezza" indispensabile durante la fase di riscrittura, evitando l'accumulo di debito tecnico, oltre che in vista di un futuro lavorativo.

Come suggerimento per il futuro, riterrei molto utile affiancare alla teoria dei singoli pattern qualche laboratorio dedicato alla costruzione passo-passo di un mini-progetto completo ("full-stack" Model-View-Controller). Questo aiuterebbe a colmare il divario tra la comprensione dei singoli componenti (che il corso spiega benissimo) e la loro integrazione in un'architettura complessa, che rimane lo scoglio principale per noi studenti.

L'utilizzo dell'AI ha rappresentato un supporto didattico significativo, non come strumento di generazione automatica di codice, ma come ausilio allo studio e alla comprensione dei concetti. È stata utilizzata in modo analogo a un docente per ripetizioni, ponendo domande mirate e prompt semi-strutturati per chiarire dubbi su specifici aspetti del linguaggio Java e della progettazione software.

Concludendo, apprezzo molto la libertà concessa dai docenti nella scelta del tema. La natura "gamificata" di J-Pou è stata la leva motivazione che almeno per me, mi ha permesso di mantenere l'entusiasmo necessario per superare le difficoltà tecniche e completare il progetto. Se avessimo dovuto affrontare le stesse difficoltà tecniche e metodologiche su un dominio più tradizionale o puramente gestionale, probabilmente non avrei mantenuto lo stesso livello di coinvolgimento ed entusiasmo fino alla fine.

Appendice A

Guida utente

Il gioco si basa su un sistema di decadimento temporale continuo, che impone all'utente di monitorare costantemente le proprie risorse al fine di evitarne l'esaurimento.

Tra i vari parametri, la Salute (**Health**) svolge un ruolo cruciale, in quanto rappresenta l'unico fattore che determina il Game Over: qualora il suo valore scenda a zero, la partita termina immediatamente e il file di salvataggio viene ripristinato con i valori di default, costringendo il giocatore a ricominciare dall'inizio.

Per mantenere il Virtual Pet in salute è possibile dormire nella Bedroom o lavarsi nel Bathroom. Le azioni disponibili per gestire le altre statistiche includono mangiare nella Kitchen e curandosi nell'Infirmary.

Il gioco include inoltre un minigioco nella Game Room che richiede l'uso della tastiera: il personaggio viene controllato tramite le frecce direzionali e l'obiettivo consiste nell'accumulare il maggior numero di punti possibile evitando le bombe (palline nere). La sessione di minigioco può essere interrotta in qualsiasi momento premendo il tasto `ESC`, oppure termina automaticamente allo scadere del timer; in entrambi i casi, lo score ottenuto viene convertito in coins e restituito al giocatore. **Nota sul consumo energetico:** I minigiochi comportano un dispendio di energia per il Pou: ogni partita completata riduce la statistica *Energy* di una quantità fissa. Se il livello di energia scende sotto la soglia minima, l'accesso al minigioco verrà bloccato; sarà quindi necessario far riposare il Pou o utilizzare una pozione energetica prima di poter giocare nuovamente.

Il salvataggio dei dati avviene in modo automatico al momento dell'uscita dal gioco, che può essere effettuata sia tramite il pulsante **Settings** sia chiudendo la finestra principale tramite la 'X'.