Make an Ice Cream

Chiara Emina, Sophy Mercuri, Antonio Scharmuller, Shuyi Zhang

13 febbraio 2026

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software oggetto di questo progetto è un videogioco intitolato Make an Ice Cream. L'obiettivo dell'applicazione è quello di offrire un'esperienza di gioco, in cui l'utente assume il ruolo di un gelatiere che deve soddisfare gli ordini di una serie di clienti entro dei vincoli temporali prestabiliti.

Durante lo svolgimento del gioco, all'utente vengono presentati diversi clienti in modo sequenziale, uno alla volta, e ognuno è caratterizzato da un ordine specifico e da un timer. L'ordine di un cliente descrive la composizione del gelato desiderato, e il timer specifica il tempo limite entro il quale l'ordine deve essere completato. Al giocatore vengono inoltre forniti, su un bancone, una serie di ingredienti quali gusti di gelato, tipi di cono e topping. Il compito del giocatore è quello di selezionare gli ingredienti appropriati e comporre il gelato in modo conforme all'ordine, per poi consegnarlo al cliente.

Il gioco è strutturato in livelli a difficoltà crescente: con l'avanzare dei livelli aumentano la complessità degli ordini e il numero complessivo di clienti, e diminuisce il tempo limite di ciascun cliente. Inoltre, il sistema prevede un numero limitato di vite pari a tre per ciascun livello. Una vita viene persa nel caso in cui il giocatore consegni un gelato non conforme all'ordine richiesto, oppure quando il tempo a disposizione per un cliente scade senza che la consegna dell'ordine venga completata. Al termine delle vite disponibili, la partita si conclude con uno stato di game over.

Requisiti funzionali

- I clienti dovranno essere presentati al giocatore in modo sequenziale, uno alla volta.

- L'applicazione dovrà consentire la composizione del gelato utilizzando gli ingredienti disponibili sul bancone.
- Il giocatore dovrà avere la possibilità di confermare la consegna del gelato composto al cliente corrente, oppure annullare la composizione del gelato in corso e riprovare la preparazione dell'ordine.
- Al momento della consegna, si dovrà verificare la conformità del gelato consegnato rispetto all'ordine richiesto.
- L'applicazione dovrà gestire un numero limitato di vite pari a tre per ciascun livello, e terminare la partita con un stato di game over quando le vite esauriscono.
- L'applicazione dovrà gestire livelli di gioco a difficoltà crescente, con ordini più articolati e numero complessivo di clienti maggiore.

Requisiti non funzionali

- L'applicazione dovrà garantire un'esperienza di gioco chiara e intuitiva per l'utente.
- L'applicazione dovrà fornire un feedback immediato e rispondere in modo fluido alle azioni del giocatore.
- L'ambiente di gioco dovrà presentare in modo chiaro all'utente le informazioni rilevanti, come l'ordine del cliente, il tempo residuo e il numero di vite disponibili.
- L'applicazione dovrà fornire una progressione graduale della difficoltà tra i livelli.

1.2 Modello del Dominio

In Make an Ice Cream, il giocatore interpreta il ruolo di un gelatiere che ha il compito di servire una serie di clienti preparando loro dei gelati. A ogni cliente sono associati un ordine, che deve indicare gli ingredienti richiesti per il gelato, e un timer che limita il tempo di consegna.

Il giocatore deve avere accesso a un insieme di ingredienti per poter comporre il gelato richiesto dall'ordine, e servire quindi il cliente. Quando il gelato viene consegnato al cliente, questo verifica se l'ordine è stato soddisfatto correttamente, e in caso negativo il giocatore perde una vita. Si può

perdere una vita anche nel caso in cui il timer associato al cliente scade e il gelato richiesto non è stato consegnato.

Il gioco è organizzato in livelli, e ogni livello ha una sua difficoltà, una sequenza di clienti da servire e un numero fisso di vite. Quando tutte le vite del livello vengono esaurite, il gioco termina con uno stato di game over. Gli elementi costitutivi del problema sono sintetizzati in **??**.

Capitolo 2

Design

2.1 Architettura

L'architettura del gioco Make an Ice Cream segue il pattern architetturale MVC, nella sua declinazione "ECB". Essa è organizzata quindi in tre componenti principali: la View (Boundary), il Controller (Control) e il Model (Entity).

L'interfaccia `GameView` rappresenta la view dell'architettura MVC e il "boundary" di ECB. Si occupa di raccogliere gli input dell'utente e di visualizzare lo stato corrente del gioco, mostrando informazioni come ad esempio l'ordine del cliente e il tempo residuo. Tutte le azioni dell'utente vengono notificate al controller sotto forma di eventi.

Il controller è rappresentato dall'interfaccia `GameController`, che si occupa del coordinamento tra le interfacce `GameView` e `Game`, cioè view e model. `GameController` elabora gli input ricevuti dalla view, invoca le operazioni opportune sul model e aggiorna lo stato del gioco, notificando poi a `GameView` le modifiche da visualizzare.

Infine, il model è rappresentato dall'interfaccia `Game`, che incapsula la logica del dominio di gioco. In particolare, esso è responsabile della gestione delle entità fondamentali del dominio e delle regole di gioco, e fornisce al controller un punto di accesso allo stato e alle regole del sistema.

L'architettura MVC/ECB del gioco è inoltre supportata dal `GameEngine`, che inizializza le componenti principali e coordina il ciclo di esecuzione tramite `GameLoop`, che a sua volta aggiorna periodicamente lo stato del gioco tramite il controller. Queste componenti garantiscono l'esecuzione continua del gioco, ma non alterano i principi di disaccoppiamento tra View, Controller e Model.

Con questa architettura, la view risulta disaccoppiata da controller e model, e può essere sostituita senza dover modificare le altre due componenti. In **??** è riportato il diagramma UML architetturale.

2.2 Design dettagliato

2.2.1 Gestione Customer, Ordini e Timer (Studente 1)

Questa sezione descrive il design relativo definendo le principali responsabilità e interazioni riguardanti la gestione dei clienti (customer), degli ordini (order) e del timer che gestisce i tempi di attesa dei clienti (CustomerTimer).

Creazione dei clienti che possiedono un ordine

Problema Nel gioco ci sono diverse tipologie di clienti. Ogni cliente ha:

- Un nome (scelto casualmente da una lista)
- Una difficoltà specifica per la composizione del gelato (mappata al nome)
- Un ordine da richiedere, generato casualmente ma sensato ai parametri stabiliti
- Un callback da attaccare al timer

Responsabilità da rispettare riguardanti il cliente

- Avere una difficoltà definita e associata al suo nome
- Disponere di un attributo Order la cui composizione non sia di difficoltà maggiore a quella proporzionata dal livello
- Disponere di un attributo Customer Timer
- Callback configurato che notifica la scadenza del timer del cliente

Approccio per la soluzione Usare il pattern "FACTORY". Perchè? Ci permette incapsulare tutta la logica di creazione dei clienti in un unico punto: `CustomerFactory`. Essa utilizza internamente una classe `CustomerTemplate` per definire le configurazioni base delle diverse difficoltà di modo che se in futuro voglio andare ad aggiungere un cliente basato su una nuova difficoltà so dove andare. Separazione delle responsabilità, il controller non deve sapere

esattamente come si fabbrica un cliente. Deve solo chiedere che gli venga dato un in base alla difficoltà da lui proporzionata. Testabilità, ci permette di testare la factory indipendentemente dal reso del gioco, verificando che i clienti siano stati generati rispettando sempre le regole.

Cosa accade dentro la factory, una volta ricevuta la difficoltà:

1. Viene scelto un nome casuale (non mappato a una difficoltà minore)
2. Gli viene impostato il tempo limite definito dal livello per la difficoltà passata
3. Genera un ordine casuale: Sceglie un cono, sceglie da 1 a ... riceve un parametro di difficoltà (dal livello corrente) e restituisce un cliente con una configurazione completa.

Ci permette in modo che se domani voglio aggiungere una nuova difficoltà per il profilo di un cliente nuovo, so esattamente dove andare. Separazione delle responsabilità, perchè il controller non deve sapere come si fa un cliente, ma deve solo chiedere un cliente in base alla difficoltà che proporziona modo che in `CustomerFactory.createCustomer()` in modo che questo riceva un intero (`maxDifficulty`) che rappresenti la difficoltà del livello in cui si troverà il cliente, in base a questo si crea un cliente di difficoltà equivalente a quella passata o inferiore. Testabilità.

Gestione degli ordini e validazione del gelato

Problema Un ordine (`Order`) è un oggetto la cui complessità è determinata da tipologia del cliente che effettua l'ordine può essere composto da:

- Una lista di gusti (1 elemento min - max 3)
- Un cono (obbligatorio)
- Una lista di topping (0-2 elementi)

Seguendo le regole di dominio imposte:

- I gusti devono essere di tipo SCOOP
- I topping devono essere LIQUID o SOLID
- Massimo un topping SOLID per ordine (non ci possono essere topping solidi tra gli SCOOP)
- Non si possono aggiungere ingredienti dopo un topping solido

Responsabilità da rispettare Costruire un ordine e memorizzarlo correttamente rispettando i parametri di dominio. Verificare se un gelato composto dal giocatore soddisfa la richiesta del cliente.

Approccio per la soluzione Usare il pattern "BUILDER". Perchè? Ci favorisce nel controllo. La classe `OrderBuilder` si occupa di costruire l'ordine ad ogni passo, controllando a sua volta che tutte le regole siano rispettate, che i tipi siano corretti e che non ci siano null, e preserva l'ordine della sequenza con cui è stato costruito il gelato. Il pattern Builder favorisce la comprensione, poiché le operazioni di stanziamento sono eseguite in sequenza. `OrderImpl` che implementa l'interfaccia `Order`, proporziona la rappresentazione dell'ordine in modo immutabile, verifica che i SOLID_TOPPING non siano maggiori a uno.

La verifica dell'ordine all'interno di `OrderImpl` la preservazione della sequenza esatta degli ingredienti del Order creato ci permette attraverso il metodo `isSatisfiedBy()` di confrontare l'`IceCream` creato dal Player (passato come parametro) e l'`Order` creato in `OrderBuilder`. L'esito positivo del confronto comprende che:

- Abbiano lo stesso cono
- Abbiano gli stessi gusti, nello stesso ordine
- Abbiano gli stessi topping, nello stesso ordine

Per garantire questo, inizialmente si verifica che il gelato passato non sia di tipo null, che sia uguale il tipo di cono (cone), poi si verifica che le lunghezze di entrambe liste sia uguale e la corrispondenza di ogni flavor e topping, posizione per posizione avvantaggiandosi delle liste di tipo `Ingredient`: con hashcode ed equal ridefinito logicamente per i topping e i flavor (dallo studente 2) in modo di poter verificare facilmente le liste corrispondente al Order e all'Ice Cream.

Gestione del timer

Problema Generale Il gioco richiede che ogni cliente abbia un timer il cui tempo rappresenti il limite di "pazienza" per ricevere il gelato. Al verificarsi lo scadere del tempo il giocatore perde una vita e il cliente non è soddisfatto. Il timer deve:

- Contare alla rovescia in modo preciso
- Poter essere messo in pausa e ripreso
- Notificare quando il suo tempo è scaduto

Problema organizzativo Coordinare l'implementazione timer alla presa di decisione dagli studenti incaricati:

- Di gestire il flusso di gioco: studente 4 (avviare, fermare, mettere in pausa)
- Dell'Implementazione delle regole di gioco, specificamente decidere cosa fare allo scadere del timer: studente 3 (togliere una vita, finire la partita, ecc)

Complicando come si sarebbe gestita la scadenza, lasciando un bisogno di modellare un timer che assumesse successivamente comportamenti specifici davanti a certe situazioni senza basarsi in metodi in costruzioni o soggetti a cambi.

Approccio per la soluzione "CALLBACK" con setter. Si è implementato il timer in modo che sapesse semplicemente contare. Il timer è stato creato con l'incertezza di cosa succede esattamente alla sua terminazione. Sa solo che, allo scadere del tempo, questo dovrà eseguire un comportamento che è stato passato dall'esterno, specificamente con un setter, non nel costruttore. Questo col proposito di favorire il lavoro in parallelo, semplificando l'implementazione e dando l'autonomia ai miei compagni di decidere come proseguere o fare cambiamenti.

Approccio per la soluzione al problema generale "Observer" (minimale): Invece dell'implementazione delle interfaccia Observer e Observable, la lista di observer e i conseguenti metodi per aggiungere o rimuovere un observer della lista, o notificare tutti si è implementato uno minimale con:

- Zero interfacce (si è usato un Runnable di Java)
- Un solo observer per il timer
- Un solo metodo per impostarlo
- Una notifica che va a uno solo e non una lista

Perchè? Nel nostro gioco, ogni timer ha esattamente un osservatore. Il cliente ha un timer, e quando quel timer scade deve avvisare il cliente (o il livello, o il controller). Non serve una lista, non serve rimuovere dinamicamente, non serve notificare dieci componenti diversi.

Vantaggi concreti di questa soluzione:

- **Testabilità**: Posso testare il timer senza dover creare un controller, un livello, un cliente o una finestra Swing. Creo un timer, gli passo un callback finto che si limita a settare una variabile booleana, simulo il passare del tempo e verifico che il callback venga chiamato. Pulito, veloce, isolato.

- **Flessibilità**: Lo stesso identico timer può essere usato per diverse parti (Clienti normali, Clienti speciali, Tutorial, Modalità allenamento). Cambia solo il callback, il timer è sempre lo stesso.

2.2.2 Costruzione del Gelato (Studente 2)

Costruzione del gelato

Problema Il sistema deve creare e gestire istanze di gelato in due situazioni diverse:

- Generare una preview del gelato richiesto dal cliente a partire dall'Order;

- Gestire il gelato costruito dal giocatore in modo incrementale (scelta cono + aggiunta ingredienti).

In entrambi i casi è necessario rispettare gli stessi vincoli di dominio e di gioco (ad esempio numero massimo di gusti, disponibilità di ingredienti legata al livello, combinazioni ammissibili). Senza una soluzione dedicata, la logica di costruzione rischia di essere duplicata tra ordine e interazione utente, rendendo più difficile mantenere coerenza ed evolvere le regole.

Soluzione La costruzione del gelato viene centralizzata in `IceCreamBuilder`, come mostrato in **??**. Il builder incapsula:

- La composizione di un gelato come combinazione di Cone e lista Ingredient;

- L'applicazione dei vincoli durante la creazione (o durante gli aggiornamenti incrementali).

L'Order non costituisce direttamente oggetti complessi, che delega al builder la creazione del gelato di preview del cliente. In modo analogo, la costruzione interattiva del giocatore utilizza lo stesso builder per aggiornare coerentemente lo stato coerente. Questa scelta riduce duplicazione, garantisce uniformità delle regole e semplifica l'introduzione di nuovi vincoli o ingredienti. Abbiamo usato in questo caso il pattern Builder, per centralizzare la logica di

costruzione e rendere riusabile lo stesso processo sia nella generazione della preview dell'ordine sia nella costruzione incrementale guidata dall'utente.

Modellazione degli ingredienti del gelato

Problema Il gelato può includere ingredienti differenti (palline di gusto, topping liquidi, topping solidi). Il sistema deve rappresentarli in modo uniforme e tipizzato, mantenendo l'estendibilità e prevenendo l'uso di valori non validi (es. stringhe arbitrarie per gusti o topping).

Soluzione Gli ingredienti condividono l'astrazione `Ingredient`, con implementazioni concrete per le diverse categorie. Inoltre, gusti e tipologie di topping sono rappresentati tramite enumerazioni dedicate, vincolando i valori ammessi dal dominio e mantenendo il modello consistente. Il gelato e il builder interagiscono esclusivamente con l'astrazione `Ingredient`, permettendo di gestire in modo polimorfico tutti i componenti del gelato e di aggiungere nuove tipologie senza modificare la struttura esistente. Questa scelta migliora l'estendibilità del modello e riduce l'accoppiamento tra le componenti.

Costruzione progressiva

Problema Il gelato è composto da diversi ingredienti variabili (base/cono, gusto, topping). Preparare tutti gli elementi contemporaneamente risulta scomodo e poco flessibile. Un approccio più naturale consiste nel prepararlo in modo progressivo, aggiungendo un ingrediente alla volta, proprio come in una gelateria (tramite click del giocatore). Alcuni ingredienti seguono regole precise: i gusti possono essere aggiunti più volte; i topping liquidi possono essere aggiunti tra due gusti; mentre i topping solidi possono essere aggiunti solo una volta ed esclusivamente sopra, una volta; l'aggiunta di un topping solido rappresenta una stato terminale che chiude il gelato e impedisce ulteriori modifiche; inoltre i topping vengono disabilitati nei primi livelli; una base può contenere al massimo tre palline. (Inoltre, per ogni cliente il gioco deve generare un gelato che chiamiamo "target" a partire dall'ordine e confrontarlo con il gelato effettivamente costruito dal giocatore tramite click.)

Soluzione Per la costruzione del gelato viene utilizzato il Builder Pattern. Il gelato è un oggetto composto da più parti (cono, sequenza di gusti, topping) e viene creato in modo progressivo, un ingrediente alla volta. Il builder centralizza la logica di costruzione e applica in un unico punto i vincoli del gioco, ad esempio: massimo tre palline, disponibilità dei topping in base al

livello, e chiusura del gelato dopo l'aggiunta di un topping solido (dopo la quale non è più possibile modificare il gelato). Questa scelta rende il modello più chiaro e riduce il rischio di duplicare le regole di controller e view, mantenendo il processo di creazione ordinato e facilmente estendibile. Il builder viene utilizzato in due contesti diversi ma con la stessa logica, garantendo coerenza tra ciò che richiede il cliente e ciò che costruisce il giocatore.

- **Gelato target (cliente)**: l'ordine del cliente fornisce una ricetta composta da tipo di cono, lista di gusti e lista di topping. Il builder interpreta questa ricetta e conosce un'istanza di `IceCream` che rappresenta il gelato target da imitare. In questo modo la figura del cliente è ottenuta applicando la stessa logica e gli stessi vincoli previsti dal gioco.

- **Gelato del giocatore**: il gelato del giocatore viene costruito progressivamente tramite click sugli ingredienti. Ogni click corrisponde a una richiesta di aggiunta (gusto o topping) gestita dal builder, che verifica i vincoli e aggiorna lo stato del gelato. Se viene aggiunto un topping solido, il gelato entra nello stato terminale di chiusura e non sono più consentite ulteriori aggiunte.

Al termine, il gelato del giocatore viene confrontato con il gelato target verificando che la configurazione finale coincida, determinando l'esito dell'ordine. Una possibile alternativa sarebbe stata gestire la costruzione del gelato come semplice modifica di liste direttamente nel controller o nella view (ad esempio aggiungendo/rimuovendo elementi e controllando manualmente i vincoli). Questa soluzione è più immediata, ma presenta lo svantaggio di distribuire la logica in più punti del codice, rendendo più probabili incoerenze e rendendo più difficile il testing. Un'ulteriore alternativa sarebbe stata adottare il Decorator Pattern, modellando ogni ingrediente come uno strato che avvolge lo stato precedente. Sebbene questa soluzione rappresenti bene l'idea di stratificazione, nel contesto del gioco risulta meno pratica per il confronto tra gelato target e gelato del giocatore e per la gestione di regole legate al livello. Inoltre, introduce un numero maggiore di classi e una struttura più complessa.

2.2.3 Gestione del Giocatore e Livelli (Studente 3)

Gestione del giocatore e della consegna degli ordini

Problema Nel gioco Make an Ice Cream il giocatore è l'entità che interagisce attivamente con il sistema: costruisce un gelato e lo consegna al cliente corrente. Il sistema deve:

- Rappresentare il giocatore come entità autonoma del modello.
- Gestire l'azione di consegna del gelato.
- Produrre un esito dell'ordine che possa influenzare lo stato del livello (avanzamento o perdita di una vita).

Soluzione Il giocatore è modellato nella classe `Player`, che incapsula le azioni principali disponibili durante il gioco. In particolare, il metodo `deliverIceCream(...)` rappresenta la consegna del gelato al cliente corrente. Il `Player` non gestisce direttamente le regole di gioco (vite, progressione o clienti), ma si limita a:

- Ricevere il gelato costruito dal giocatore.
- Inoltrare la consegna al cliente corrente.
- Ricevere l'esito dell'operazione dal cliente.

Il livello utilizza tale esito per aggiornare il proprio stato, consentendo una separazione netta tra l'azione del giocatore (`Player`) e le regole del gioco (`Level`). Questa scelta mantiene il modello modulare e favorisce il riuso del codice, evitando che la logica della consegna venga duplicata o sparsa in componenti non appropriati.**??**.

Gestione dei livelli e progressione della difficoltà

Problema Il gioco Make an Ice Cream è strutturato in livelli a difficoltà crescente. Prima dell'inizio della partita il giocatore seleziona un livello, ma il sistema deve:

- Creare un livello coerente con la difficoltà scelta.
- Inizializzare correttamente vite, numero di clienti e parametri di gioco.
- Garantire una progressione graduale della difficoltà.
- Mantenere la logica di creazione e gestione del livello separata dal controller e dalla view.

Soluzione Per la gestione dei livelli è stato utilizzato il pattern *Simple Factory*. Prima dell'inizio del livello il giocatore seleziona un valore intero compreso tra 1 e 5, che rappresenta il livello scelto. Questo valore viene poi utilizzato come parametro di difficoltà passato alla factory per la creazione del livello. Il livello creato gestisce le vite disponibili e la sequenza di clienti, a cui è stato propagato il parametro di difficoltà che influenza la complessità degli ordini e i tempi di attesa dei clienti, garantendo una progressione graduale di difficoltà. L'uso del factory consente di centralizzare la logica per poi evitare duplicazioni necessarie future per estendere il gioco introducendo nuove tipologie di livelli.**??**.

2.2.4 Gestione Comandi e Stato (Studente 4)

Gestione dei comandi utente

Problema Il controller del gioco deve gestire tutte le possibili azioni dell'utente, e il sistema deve permettere di aggiungere nuovi comandi in futuro senza modificare il controller esistente.

Soluzione Il sistema per la gestione dei comandi utilizza il pattern Command, come da **??**: ogni azione dell'utente è rappresentata da un oggetto `Command` che incapsula la logica specifica dell'azione e delega l'esecuzione di essa al modello `Game`, che funge da receiver. Il controller, ovvero `GameControllerImpl`, agisce invece da invoker, creando ed eseguendo il comando corrispondente all'azione ricevuta, senza però saperne l'implementazione. In questo modo, nuove azioni possono essere aggiunte creando semplicemente un nuovo comando e mantenendo il controller semplice e leggibile. A titolo esemplificativo, in **??**, `AddIngredientCommand` e `DeliverCommand` mostrano come un comando concreto implementa l'interfaccia `Command` e interagisce con il modello.

Gestione degli eventi utente

Problema Il sistema deve rappresentare in modo uniforme gli eventi generati dalle azioni dell'utente sull'interfaccia grafica, per permettere al controller di gestirle correttamente senza conoscere i dettagli interni della view.

Soluzione Ogni evento è rappresentato da un oggetto `EventImpl`, che implementa l'interfaccia `Event` e associa l'azione a un valore dell'enumerazione `EventType`, come mostrato in **??**. In questo modo il controller può interrogare il tipo di azione e creare il comando corrispondente senza preoccuparsi

della logica interna. Questo design consente al controller di gestire tutti gli eventi in modo uniforme e può essere facilmente esteso per supportare nuovi tipi di eventi.

Gestione dello stato del gioco

Problema Il sistema deve tenere traccia dello stato corrente del gioco e gestire correttamente transizioni come avvio di un livello, pausa e ripresa, fine livello o game over.

Soluzione Come mostrato in **??**, `GameImpl` utilizza l'enumerazione `GameState` per rappresentare lo stato corrente del gioco. Ogni metodo (`start()`, `pause()`, `resume()`, `returnToMenu()`) modifica lo stato in modo appropriato, mentre `update()` verifica le condizioni di avanzamento del livello e aggiorna lo stato in base al numero di vite o ai clienti serviti. La scelta di usare un'enumerazione è motivata dalla semplicità del dominio: non vi è un numero elevato di stati ed essi non hanno una complessità tale da dover utilizzare un pattern State, che sarebbe eccessivo per questo caso d'uso.

Motore di gioco

Problema Il gioco deve aggiornare periodicamente lo stato del modello e notificare la view, senza mescolare logica di esecuzione e gestione degli eventi.

Soluzione Il `GameEngine` inizializza il modello (`GameImpl`) e il controller (`GameControllerImpl`), e coordina il ciclo principale di esecuzione tramite `GameLoop`. Il loop calcola il tempo trascorso tra un frame e l'altro e chiama il controller per aggiornare il modello solo se il gioco è in esecuzione. In questo modo la logica del modello e del controller rimane separata dalla gestione del tempo, facilitando estensioni future.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Player: sono stati testati la costruzione del gelato in stati validi, la gestione di input non validi e il corretto comportamento della consegna verso il cliente. Mediante oggetti mock è stato verificato l'esito positivo e negativo dell'ordine, controllando anche il reset dello stato interno dopo ogni consegna. È stato inoltre testato l'annullamento della costruzione e la gestione di stati non validi.

Level (StandardLevel):è stato testato lo stato iniziale del livello (vite, difficoltà, presenza clienti) e la corretta progressione della coda. Sono state verificate la perdita di vite in caso di ordine errato o tempo scaduto e l'aggiornamento temporale tramite timer mockato. È stata inoltre verificata la registrazione e gestione delle callback per propagare l'esito dell'ordine al livello.

Elementi positivi

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

Elementi negativi

- Non si realizza alcun test automatico.

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

3.2 Note di sviluppo

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

- **Elencare** (fare un semplice elenco per punti, non un testo!) le feature *avanzate* del linguaggio e dell'ecosistema Java che sono state utilizzate. Le feature di interesse sono:
  - Progettazione con generici, ad esempio costruzione di nuovi tipi generici, e uso di generici bounded. L'uso di classi generiche di libreria non è considerato avanzato.
  - Uso di lambda expressions
  - Uso di `Stream`, di `Optional` o di altri costrutti funzionali
  - Uso di reflection
  - Definizione ed uso di nuove annotazioni
  - Uso del Java Platform Module System
  - Uso di parti della libreria JDK non spiegate a lezione (networking, compressione, parsing XML, eccetera...)

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

  - Uso di librerie di terze parti (incluso JavaFX): Google Guava, Apache Commons...

- Si faccia molta attenzione a non scrivere banalità, elencando qui features di tipo "core", come le eccezioni, le enumerazioni, o le inner class: nessuna di queste è considerata avanzata.

- Per ogni feature avanzata, mostrata, includere:

  - Nome della feature
  - Permalink GitHub al punto nel codice in cui è stata utilizzata

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

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

Elementi positivi

- Si elencano gli aspetti avanzati di linguaggio che sono stati impiegati
- Si elencano le librerie che sono state utilizzate
- Per ciascun elemento, si fornisce un permalink
- Ogni permalink fa riferimento ad uno snippet di codice scritto dall'autore della sezione (i docenti verificheranno usando `git blame`)
- Se si è utilizzato un particolare algoritmo, se ne cita la fonte originale. Ad esempio, se si è usato Mersenne Twister per la generazione di numeri pseudo-random, si cita [**?**].

- Si identificano parti di codice prese da altri progetti, dal web, o comunque scritte in forma originale da altre persone. In tal senso, si ricorda che agli ingegneri non è richiesto di re-inventare la ruota continuamente: se si cita debitamente la sorgente è tollerato fare uso di di snippet di codice open source per risolvere velocemente problemi non banali. Nel caso in cui si usino snippet di codice di qualità discutibile, oltre a menzionarne l'autore originale si invitano gli studenti ad adeguare tali parti di codice agli standard e allo stile del progetto. Contestualmente, si fa presente che è largamente meglio fare uso di una libreria che copiarsi pezzi di codice: qualora vi sia scelta (e tipicamente c'è), si preferisca la prima via.

Elementi negativi

- Si elencano feature core del linguaggio invece di quelle segnalate. Esempi di feature core da non menzionare sono:
  - eccezioni;
  - classi innestate;
  - enumerazioni;
  - interfacce.
- Si elencano applicazioni di terze parti (peggio se per usarle occorre licenza, e lo studente ne è sprovvisto) che non c'entrano nulla con lo sviluppo, ad esempio:
  - Editor di grafica vettoriale come Inkscape o Adobe Illustrator;
  - Editor di grafica scalare come GIMP o Adobe Photoshop;
  - Editor di audio come Audacity;
  - Strumenti di design dell'interfaccia grafica come SceneBuilder: il codice è in ogni caso inteso come sviluppato da voi.
- Si descrivono aspetti di scarsa rilevanza, o si scende in dettagli inutili.
- Sono presenti parti di codice sviluppate originalmente da altri che non vengono debitamente segnalate. In tal senso, si ricorda agli studenti che i docenti hanno accesso a tutti i progetti degli anni passati, a Stack Overflow, ai principali blog di sviluppatori ed esperti Java, ai blog dedicati allo sviluppo di soluzioni e applicazioni (inclusi blog dedicati ad Android e allo sviluppo di videogame), nonché ai vari GitHub, GitLab,

e Bitbucket. Conseguentemente, è *molto* conveniente *citare* una fonte ed usarla invece di tentare di spacciare per proprio il lavoro di altri.

- Si elencano design pattern

- **Utilizzo di Lambda expressions**
  Le lambda expressions sono state utilizzate per registrare dinamicamente le callback tra cliente e livello, permettendo di propagare l'esito dell'ordine in modo compatto e leggibile. L'uso di funzioni come parametro riduce la necessità di classi anonime esplicite e migliora la chiarezza del flusso logico.
  Link: `[PERMALINKStandardLevel--registrazionecallback]`

- **Utilizzo di strutture dati della Collections Framework (Queue / ArrayDeque)**
  La gestione dei clienti nel livello avviene tramite `Queue` e `ArrayDeque`, garantendo un comportamento FIFO, consente una gestione efficiente dell'avanzamento del gioco e rende il modello più prevedibile nei test.
  Link: `[PERMALINKStandardLevel--dichiarazioneeutilizzoQueue]`

- **Utilizzo di Mockito per il testing**
  Nei test automatici è stata utilizzata la libreria Mockito per creare mock di `Customer` e `Timer`. Questo ha permesso di isolare il comportamento e verificare effetti collaterali e simulare condizioni specifiche (es. scadenza del tempo).
  Link: `[PERMALINKPlayerTest/StandardLevelTest--utilizzomockewhen/verify]`

- **Controllo dello stato tramite validazione esplicita**
  Sono stati introdotti controlli espliciti su stati non validi (es. consegna senza gelato completo) e su input nulli. Questa scelta migliora la robustezza del modello e previene stati inconsistenti durante l'esecuzione.
  Link: `[PERMALINKPlayerImpl--controllisustatoenull]`

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

Capitolo in cui si spiega come utilizzare il software. Nel caso in cui il suo uso sia del tutto banale, tale capitolo può essere omesso. A tal riguardo, si fa presente agli studenti che i docenti non hanno mai utilizzato il software prima, per cui aspetti che sembrano del tutto banali a chi ha sviluppato l'applicazione possono non esserlo per chi la usa per la prima volta. Se, ad esempio, per cominciare una partita con un videogioco è necessario premere la barra spaziatrice, o il tasto "P", è necessario che gli studenti lo segnalino.

Elementi positivi

- Si istruisce in modo semplice l'utente sull'uso dell'applicazione, eventualmente facendo uso di schermate e descrizioni.

Elementi negativi

- Si descrivono in modo eccessivamente minuzioso tutte le caratteristiche, anche minori, del software in oggetto.
- Manca una descrizione che consenta ad un utente qualunque di utilizzare almeno le funzionalità primarie dell'applicativo.

Appendice B

Esercitazioni di laboratorio

In questo capitolo ciascuno studente elenca gli esercizi di laboratorio che ha svolto (se ne ha svolti), elencando i permalink dei post sul forum dove è avvenuta la consegna. Questa sezione potrebbe essere processata da strumenti automatici, per cui link a oggetti diversi dal permalink della consegna, errori nell'email o nel nome del laboratorio possono portare ad ignorare alcune consegne, si raccomanda la massima precisione.

B.0.1 Studente 1

B.0.2 Studente 2

B.0.3 Studente 3

B.0.4 chiara.emina@studio.unibo.it

- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=207193#p284987`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=207921#p286132`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=208718#p286856`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=209589#p288534`
- Laboratorio 11: `https://virtuale.unibo.it/mod/forum/discuss.php?d=210617#p289525`