Relazione Temple-Tower per "Programmazione ad Oggetti"

Davide Vignali, Marko Cobo, Mattia Mularoni, Nicolas Montanari

30 dicembre 2024

Indice

1 Analisi **2**
1.1 Descrizione e requisiti 2
1.2 Modello del Dominio 3

2 Design **6**
2.1 Architettura 6
2.2 Design dettagliato 7

3 Sviluppo **16**
3.1 Testing automatizzato 16
3.2 Note di sviluppo 17
3.2.1 Esempio 19

4 Commenti finali **21**
4.1 Autovalutazione e lavori futuri 21
4.2 Difficoltà incontrate e commenti per i docenti 21

A Guida utente **23**

B Esercitazioni di laboratorio **24**
B.0.1 paolino.paperino@studio.unibo.it 24
B.0.2 paperon.depaperoni@studio.unibo.it 24

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il progetto Temple Tower si ispira ai classici dungeon crawler, offrendo un'esperienza di gioco a livelli in cui il giocatore esplora piani di un dungeon circolare per raccogliere tesori, sconfiggere nemici e salire verso il livello successivo. Il gioco culmina in un epico scontro con un boss finale. La meccanica dei livelli circolari prende ispirazione dal gioco Ring of Pain.

Requisiti funzionali

- L'area di gioco è circolare ed è composta da diverse caselle contententi gli elementi di gioco che possono essere positivi o negativi per il giocatore.
- Gli elementi di gioco sono rappresentati da tesori (possono contenere punti esperienza, armi, oggetti curativi), trappole (tolgono punti vita), scale (permettono il passaggio a un livello superiore).
- Il giocatore si può muovere all'interno del livello, può combattere contro i nemici, interagire con gli elementi di gioco.
- Dopo ogni scontro con un nemico o dopo aver attraversato una trappola il giocatore perde i punti vita, al termine dei quali la partita termina e si ritorna alla schermata inziale. È possibile riacquistare punti vita attraverso gli oggetti curativi.
- Generazione casuale della torre per permettere partite sempre diverse.

Requisiti non funzionali

- Una volta arrivato all'ultimo piano il giocatore incontrerà il boss finale, il quale avrà comportamenti unici e una difficoltà maggiore rispetto ai nemici normali .
- Difficoltà variabile in base al progresso di gioco.
- Musica di sottofondo e feedback sonori associati alle varie azioni.
- Salvataggio risultati delle partite precedenti, visualizzabili in un'apposita schermata.
- Esistono più tipologie di attacchi la quale efficacia varia rispetto al tipo di nemico.

1.2 Modello del Dominio

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

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

Elementi positivi

- Si modella il dominio in forma di UML.
- Viene descritto accuratamente il modello del dominio.

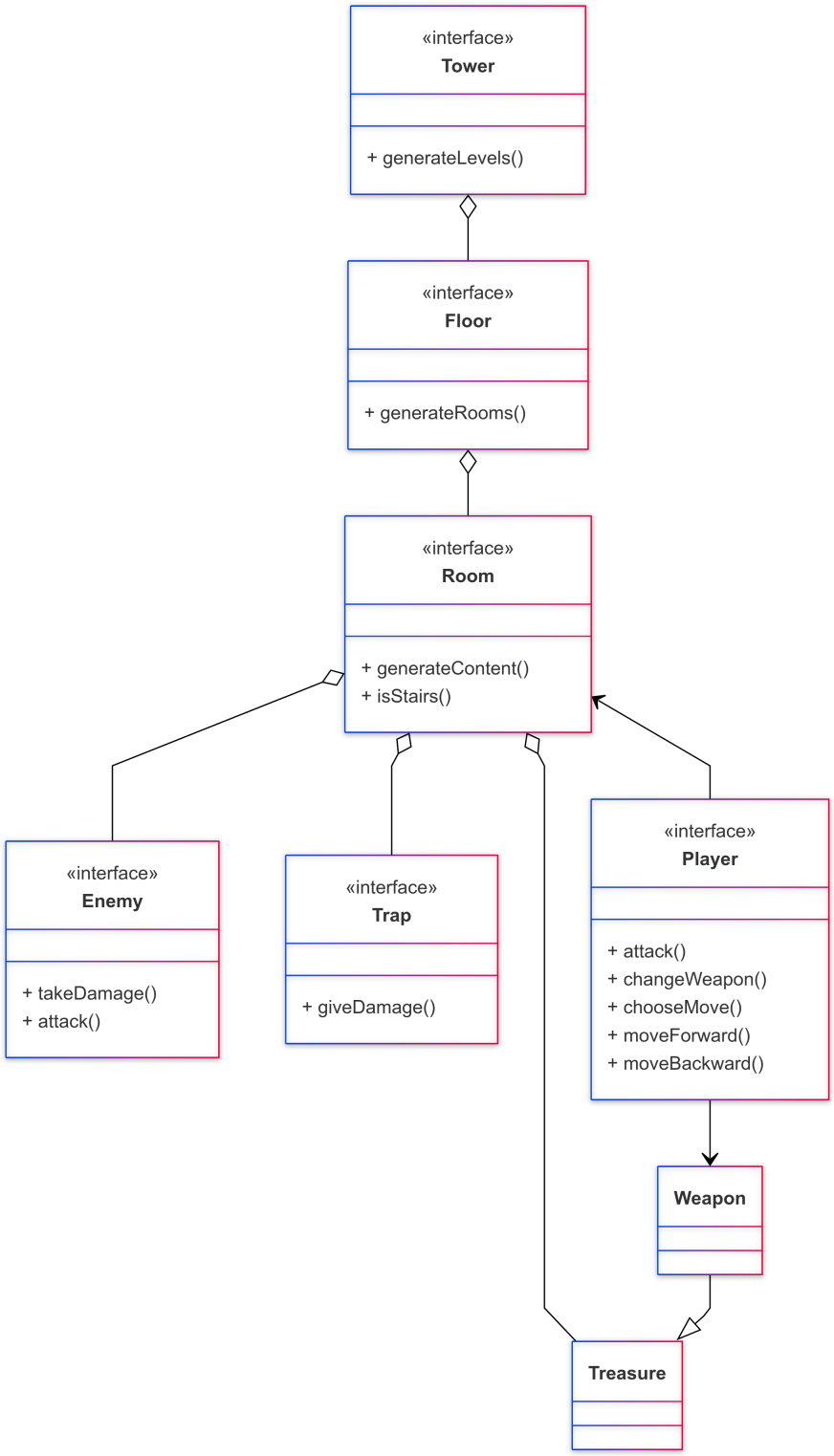
- Il modello cattura le entità di dominio, le loro caratteristiche principali,
- e le relazioni che intercorrono fra di loro.

Elementi negativi

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

Esempio

Temple Tower dovrà essere rappresentato da una torre formata da un certo numero di piani (Floor), contenenti delle stanze collegate tra loro, ogni stanza potrà contentere una trappola, un nemico o un tesoro e il giocatore che va al suo interno. Una trappola può arrecare danno al giocatore (come il nemico) o al contrario, un tesoro puà fornirgli punti vita/esperienza (oppure un'arma). Il giocatore utilizza armi con le quali combattere: in quanto tutti gli attacchi avverranno a turno, bisognerà individuare una strategia per ridurre le vittorie "matematiche" dovute alla differenza di efficacia. Con le armi a disposizione, il giocatore e il nemico avranno a disposizione varie tipologie di mosse di attacco. Il giocatore dopo aver oltrepassato le varie stanze, potrà utilizzare le scale per passare al piano successivo della torre. Durante tutta la durata della partita, il giocatore e i nemici avranno due barre di stato, una rappresentante la vita e l'altra l'esperienza: ad ogni scaglione di esperienza, si sbloccheranno certe abilità o miglioramenti, finita la disponibilità dei punti vita, i quali vengono sottratti dai nemici e dalle trappole si perde la partita e si ricomincia da capo. Arrivati all'ultimo livello, si affronterà il boss finale, che è un nemico più forte con mosse di attacco particolari e più avanzate dei nemici generici.


«interface»
Tower
+ generateLevels()
«interface»
Floor
+ generateRooms()
«interface»
Room
+ generateContent()
+ isStairs()
«interface»
Enemy
+ takeDamage()
+ attack()
«interface»
Trap
+ giveDamage()
«interface»
Player
+ attack()
+ changeWeapon()
+ chooseMove()
+ moveForward()
+ moveBackward()
Weapon
Treasure

Capitolo 2

Design

2.1 Architettura

L'architettura del gioco **Temple Tower** segue il pattern architetturale **Model-View-Controller (MVC)** per garantire una chiara separazione delle responsabilità tra la logica di business, la presentazione e la gestione degli eventi.

Composizione del Pattern MVC

- **Model**:
  - Rappresenta la logica principale del gioco e include classi come `Tower`, `Floor`, `RoomBehavior`, `Player`, e i vari tipi di stanze (`EnemyRoom`, `TreasureRoom`, `StairsRoom`) che implementano il pattern Strategy.
  - Questo approccio consente di definire comportamenti specifici per ogni tipologia di stanza in modo modulare, rendendo semplice l'aggiunta di nuovi tipi di stanze senza modificare il codice esistente.
  - La logica di gioco, come il movimento del giocatore o gli effetti delle interazioni con nemici, trappole o tesori, è interamente contenuta nel model.
- **View**:
  - La classe `GameView` gestisce la presentazione grafica del gioco.

  - È responsabile della visualizzazione dello stato attuale della stanza (`displayRoom`), degli aggiornamenti della barra di stato del giocatore (`updateStatusBar`) e della gestione di eventi speciali come la vittoria o il game over.

- **Controller**:
  - La classe `GameController` funge da intermediario tra il modello e la vista, orchestrando il flusso degli eventi nel gioco.
  - Gestisce le azioni dell'utente (`handleAction`), permette al giocatore di muoversi tra le stanze (`changeRoom`) o salire al piano successivo (`gotoNextFloor`), e si occupa di iniziare e terminare il gioco.

Scalabilità e Manutenibilità

Grazie all'uso combinato dei pattern MVC e Strategy:

- **Aggiunta di nuove stanze**: È possibile introdurre nuove tipologie di stanze semplicemente aggiungendo nuove implementazioni dell'interfaccia `RoomBehavior`, senza modificare altre parti del codice.
- **Separazione delle responsabilità**: La gestione della logica di gioco, della presentazione grafica e delle interazioni dell'utente è ben separata, favorendo la manutenibilità e la possibilità di cambiare singole componenti senza influenzare le altre.

Questa architettura rende il sistema flessibile, modulare e facilmente estensibile, adattandosi alle necessità di futuri miglioramenti o aggiunte.

2.2 Design dettagliato

In questa sezione si possono approfondire alcuni elementi del design con maggior dettaglio. Mentre ci attendiamo principalmente (o solo) interfacce negli schemi UML delle sezioni precedenti, in questa sezione è necessario scendere in maggior dettaglio presentando la struttura di alcune sottoparti rilevanti dell'applicazione. È molto importante che, descrivendo la soluzione ad un problema, quando possibile si mostri che non si è re-inventata la ruota ma si è applicato un design pattern noto. Che si sia utilizzato (o riconosciuto) o meno un pattern noto, è comunque bene definire qual è il problema che si è affrontato, qual è la soluzione messa in campo, e quali motivazioni l'hanno spinta. È assolutamente inutile, ed è anzi controproducente, descrivere

classe-per-classe (o peggio ancora metodo-per-metodo) com'è fatto il vostro software: è un livello di dettaglio proprio della documentazione dell'API (deducibile dalla Javadoc).

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

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

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

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

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

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

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

Elementi positivi

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

Elementi negativi

- Il design del modello risulta scorrelato dal problema descritto in analisi.
- Si tratta in modo prolisso, classe per classe, il software realizzato, o comunque si riduce la sezione ad un mero elenco di quanto fatto.
- Non si presentano schemi UML esemplificativi.
- Non si individuano design pattern, o si individuano in modo errato (si spaccia per design pattern qualcosa che non lo è).

- Si utilizzano design pattern in modo inopportuno. Un esempio classico è l'abuso di Singleton per entità che possono essere univoche ma non devono necessariamente esserlo. Si rammenta che Singleton ha senso nel secondo caso (ad esempio `System` e `Runtime` sono singleton), mentre rischia di essere un problema nel secondo. Ad esempio, se si rendesse singleton il motore di un videogioco, sarebbe impossibile riusarlo per costruire un server per partite online (dove, presumibilmente, si gestiscono parallelamente più partite).

- Si producono schemi UML caotici e difficili da leggere, che comprendono inutili elementi di dettaglio.

- Si presentano schemi UML con classi (nel senso UML del termine) che "galleggiano" nello schema, non connesse, ossia senza relazioni con il resto degli elementi inseriti.

- Si tratta in modo inutilmente prolisso la divisione in package, elencando ad esempio le classi una per una.

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

Personalità intercambiabili

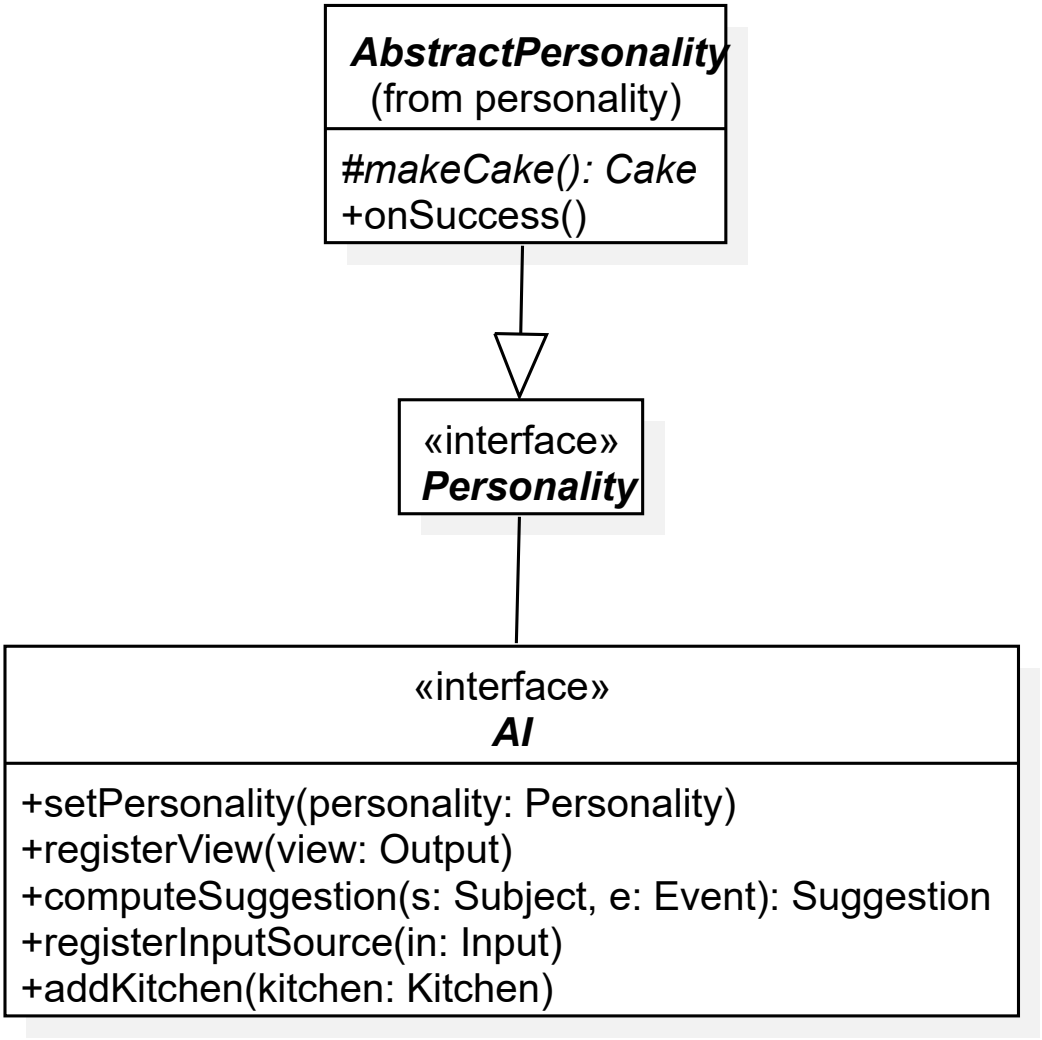


Figura 2.1: Rappresentazione UML del pattern Strategy per la personalità di GLaDOS

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

Soluzione Il sistema per la gestione della personalità utilizza il *pattern Strategy*, come da Figura 2.1: le implementazioni di `Personality` possono

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

Riuso del codice delle personalità

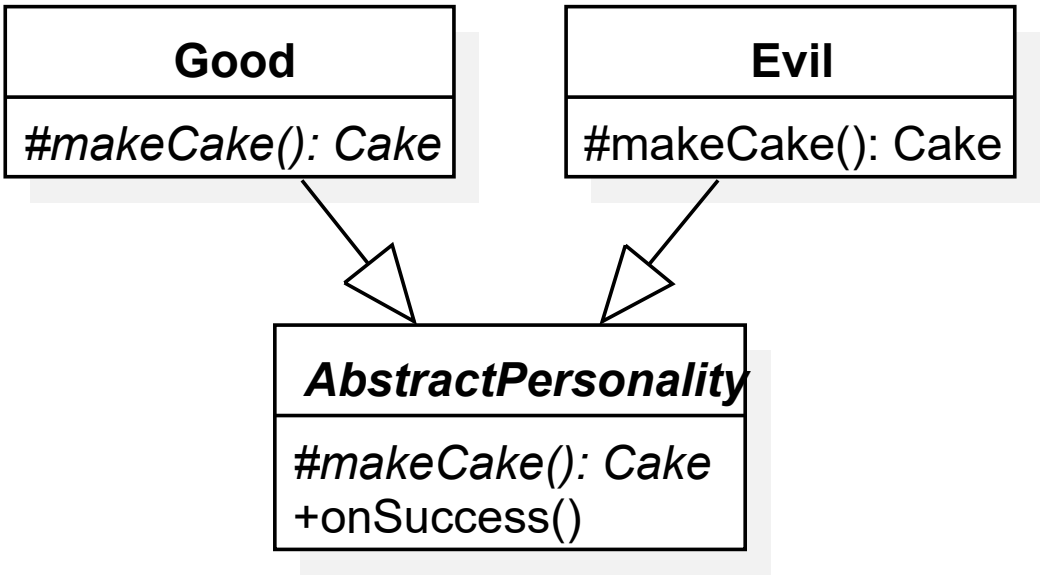


Figura 2.2: Rappresentazione UML dell'applicazione del pattern Template Method alla gerarchia delle Personalità

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

Soluzione Dato che le due personalità differiscono solo per il comportamento da effettuarsi in caso di percorso completato con successo, è stato utilizzato il *pattern template method* per massimizzare il riuso, come da Figura 2.2. Il metodo template è `onSuccess()`, che chiama un metodo astratto e protetto `makeCake()`.

Gestione di output multipli

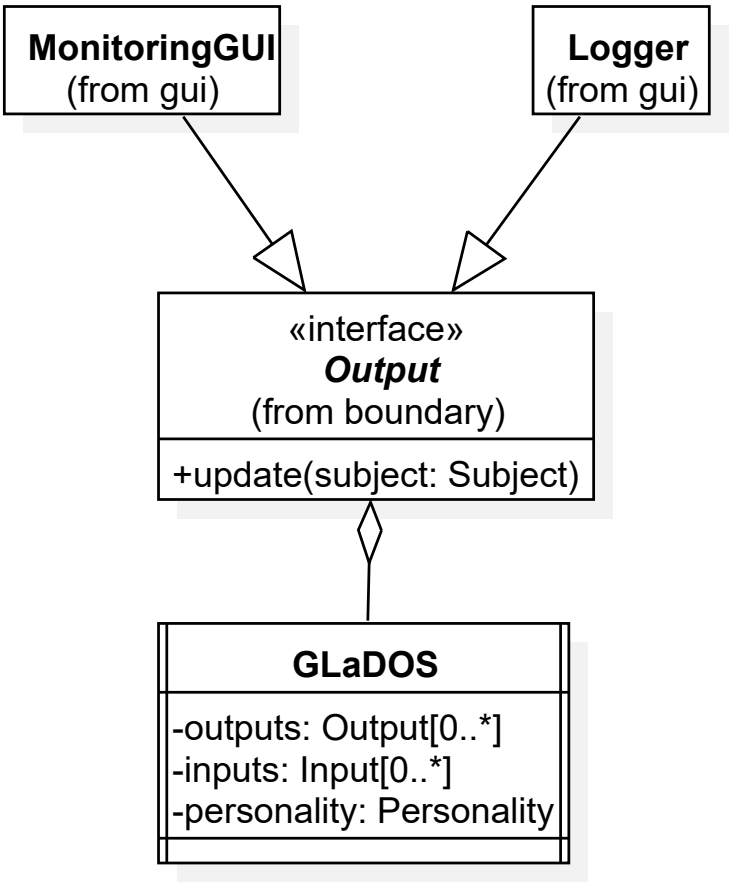


Figura 2.3: Il pattern Observer è usato per consentire a GLaDOS di informare tutti i sistemi di output in ascolto

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

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

observable, e le istanze di `Output` sono observer. Il suo utilizzo è esemplificato in Figura 2.3

Contro-esempio: pessimo diagramma UML

In Figura 2.4 è mostrato il modo **sbagliato** di fare le cose. Questo schema è fatto male perché:

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

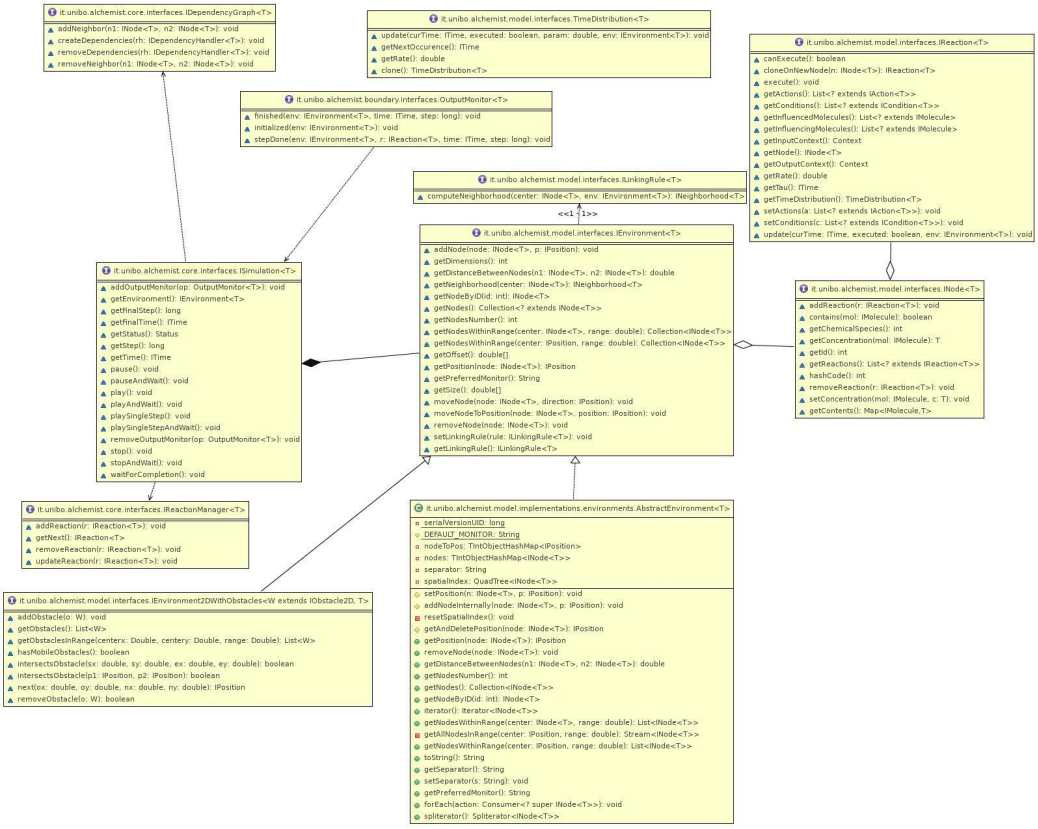


Figura 2.4: Schema UML mal fatto e con una pessima descrizione, che non aiuta a capire. Don't try this at home.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il testing automatizzato è un requisito di qualunque progetto software che si rispetti, e consente di verificare che non vi siano regressioni nelle funzionalità a fronte di aggiornamenti. Per quanto riguarda questo progetto è considerato sufficiente un test minimale, a patto che sia completamente automatico. Test che richiedono l'intervento da parte dell'utente sono considerati *negativamente* nel computo del punteggio finale.

Elementi positivi

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

Elementi negativi

- Non si realizza alcun test automatico.
- La non presenza di testing viene aggravata dall'adduzione di motivazioni non valide. Ad esempio, si scrive che l'interfaccia grafica non è testata automaticamente perché è *impossibile* farlo[1].
- Si descrive un testing di tipo manuale in maniera prolissa.

[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!

- 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...)
  - 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

3.2.1 Esempio

Utilizzo della libreria SLF4J

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

Utilizzo di `LoadingCache` dalla libreria Google Guava

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

Utilizzo di `Stream` e lambda expressions

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

Scrittura di metodo generico con parametri contravarianti

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

Protezione da corse critiche usando `Semaphore`

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

Capitolo 4

Commenti finali

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

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

4.1 Autovalutazione e lavori futuri

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.

Esempio

B.0.1 paolino.paperino@studio.unibo.it

- Laboratorio 04: `https://virtuale.unibo.it/mod/forum/discuss.php?d=12345#p123456`
- Laboratorio 06: `https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999`

B.0.2 paperon.depaperoni@studio.unibo.it

- Laboratorio 04: `https://virtuale.unibo.it/mod/forum/discuss.php?d=12345#p123456`
- Laboratorio 05: `https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222`

- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222