Capitolo 1

Analisi

Il gioco SpaceSurvival nasce come rivisitazione del famoso arcade Asteroids, l’obiettivo è creare un gioco semplice nel quale però l’utente debba crearsi una strategia per non essere sconfitto dai nemici, da qui appunto il nome.

L’utente, che ha il controllo della navicella, si pone l’obbiettivo di superare più livelli possibili prima che la sua navicella venga distrutta.

Ha la possibilità di sparare a nemici e raccogliere vita o perk che andranno a differenziare il tipo di munizione sparata.

I nemici che vengono presentati all’utente sono differenziati in nemici attivi e passivi e possiamo dividerli in tre categorie:

* Asteroids, quindi asteroidi che seguono il loro moto rettilineo.
* Chase Enemies, quid nemici che inseguono la navicella.
* Fire Enemies, quindi nemici che seguono un movimento casuale ma con un arma a disposizione e sempre puntata verso la navicella.

Il gioco ha una difficoltà incrementale data dal numero via via maggiore di nemici oltre ad un boss da sconfiggere ogni cinque livelli.

La navicella, comandata tramite la tastiera dall’utente, ha un’arma e proiettili di tipo normale che può sparare infinitamente.

La sua vita iniziale è 400 life points che può aumentare raccogliendo perks di tipo vita.

In totale i perk disponibili sono cinque, tre relativi alle munizioni sparate e due relativi alla vita della navicella.

I perk relativi alle munizioni cambiano l’effetto provocato sui nemici e sono tre:

* Proiettile fuoco: infuoca i nemici infliggendo un danno nel tempo
* Proiettile ghiaccio: rallenta la velocità del nemico.
* Proiettile elettrico: ferma definitivamente il nemico colpito.

Queste munizioni, a differenza delle classiche non sono infinite ma sono 25 per ogni tipo raccolto.

Il boss avrà a disposizione tutti i tipi di munizioni e le alternerà ogni X secondi.

Quelli relativi alla vita invece sono:

* Cuore verde: aumenta di 100 life points la vita della navicella.
* Cuore rosso: aumenta di X life points la vita della navicella.

Non vi è un limite massimo di vita e l’utente viene sconfitto quando finisce i suoi life points.

Requisiti funzionali

Permette all’utente una personalizzazione della navicella tramite il cambio di skin.

L’utente potrà impostare una difficoltà di gioco tramite la sezione “Settings”.

L’applicazione permette di gestire i suoni di background e gli effetti nel gioco in maniera separata.

Vi è una sezione “help” nella quale l’utente capisce come poter giocare e dove vengono descritti i comandi.

L’utente avrà la possibilità di abbandonare il gioco.

L’utente in partita si troverà davanti un HUD composto dalla vita corrente della navicella, il punteggio attuale, il tempo trascorso in partita, i nemici rimanenti per passare al livello successivo e il tipo di munizione corrente.

L’applicazione permette il movimento controllato della navicella all’interno del campo di gioco.

L’utente avrà la possibilità di mettere in pausa, e qui di tornare alla partita o ancora, abbandonare il gioco.

L’utente ha la possibilità anche in partita di gestire come meglio ritiene opportuno i suoni.

L’applicazione permette all’utente di svilupparsi una strategia di gioco.

Requisiti non funzionali

* Gestione delle risorse del computer in maniera quasi ottimale.
* Struttura scalabile per future implementazioni, facilità di aggiunta o rimozione oggetti di gioco.

Analisi e modello del dominio.

Il gioco si baserà su una semplice interfaccia, con una struttura che renderà facile lo scambio di singoli suoi elementi.

Sarà presente una navicella identificata dall’entità SpaceShipSigleton che dovrà essere controllata dall’utente, quest’ultima esulerà da tutti gli altri elementi di gioco.

Gli elementi di gioco che dovranno comparire dipenderanno dai livelli, e giunti al livello nel quale compare il nemico “più forte” (identificato come Boss) li avremo incontrati tutti.

Gli elementi di gioco sono

* Spaceship.
* Asteroids
* Chase Enemies
* Fire Enemies

Ma sarà importante sviluppare in maniera da rendere facile l’implementazione di nuovi oggetti di gioco, ognuno dei quali può avere macro caratteristiche in comune come il tipo di movimento o avere la possibilità di sparare o meno.

Design

L’architettura di Space Survival è stato creata seguendo il pattern di architetturale MVC, Model, View e Controller.

Model

Il model è stato implementato in modo da fornire al controller metodi corretti per l’accesso ai dati oltre a gestire come questi vengono influenzati tramite il cambiamento di stati, entità o iterazioni da parte dell’utente finale.

Il model inoltre rende corretto e in modo coerente ed è l’entità sulla quale si basa la corretta comunicazione tra il controller e la view.

View

La view è composta dalle GUI del gioco e queste ultime rimangono indipendenti dal model e dal controller del gioco.

Rimane facilmente implementabile e interscambiabile. Il compito della View è rendere visibili all’utente finale i dati del model in maniera facilmente interpretabile.

Inoltre la View ha il fondamentale compito di rendere disponibili al controller quelle che sono le iterazioni dell’utente nelle varie GUI per poi passarle al model del nostro gioco.

Controller

Il controller è quel componente dell’architettura che ha il compito di controllare e gestire l’ordine e la consequenzialità degli eventi.

Il controller inoltre deve gestire i comandi dell’utente letti tramite la View o input da tastiera per poterli trasferire al model e aspettare eventuali risposte e reindirizzandole nuovamente alla view.

Il

Controller

è quella componente cui spetta gestire in maniera conseguenziale le in-

terazioni da parte dell’utente nella

View

, comunicando quindi al

Model

il cambiamento

avvenuto.

Una volta che il

Model

avrà completato l’elaborazione della richiesta di cambiamento, il

Controller

avviserà la propria

View

, in modo tale che quest’ultima possa aggiornarsi in

10

maniera coerente secondo le regole specificate dal

Model

.

Come già accennato nella sezione precedente, si è scelto di evitare l’utilizzo di un singolo

controller, sia per i motivi sopracitati, sia perché un unico controller sarebbe risultato es-

sere una "God Class", avente un eccessivo grado di responsabilità.

Abbiamo deciso quindi di organizzare anche il lato controller secondo una struttura ge-

rarchica, dove l’entità del

Controller

è rappresentata dall’interfaccia che ne incapsula il

funzionamento, ossia avere l’accesso al

Model

e alla

View

.

Da questo derivano poi le varie specializzazioni dei vari

Controller

, ognuno specifico se-

condo il contratto delle funzionalità, determinanti queste quali parti del

Model

concernono

gli accessi del suddetto.

11

img/MVC/controller.png

Figura 2.4: Organizzazione dei

Controller

nell’architettura MVC

Quello che ne è derivato è stata un architettura dell’applicazione con un alto livello di esten-

dibilità, che ha agevolato di molto lo sviluppo, poiché ogni pagina e rispettivo

Control-

ler

potevano lavorare indipendentemente dalle altre coppie

View

-

Controller

, rendendo

possibile, e soprattutto molto semplice, integrare nuove funzionalità.

Design nello specifico

Buizo

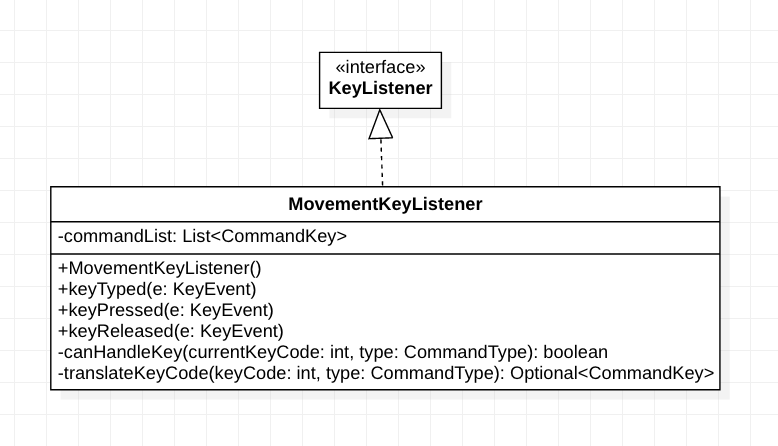
Matteini

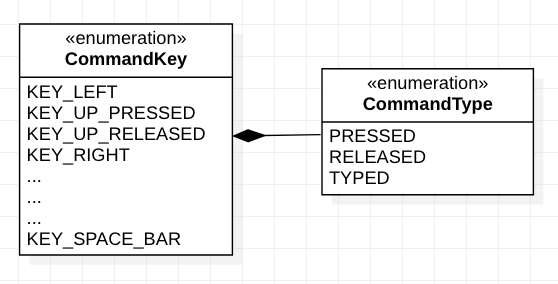
Paganelli azioni della navicella.

Un compito fondamentale per Space Survival è il movimento della navicella che viene comandata tramite tastiera dall’utente.

E’ stato creato il Command Pattern sia per i principali comandi implementati che per la comodità dello stesso in caso di necessità per comandi futuri.

Essendo tutti gli input provenienti dalla tastiera, è stato creato un MovementKeyListener personalizzato che accetta solo i tipi di comandi specificati nell’Enumeration CommandKey.





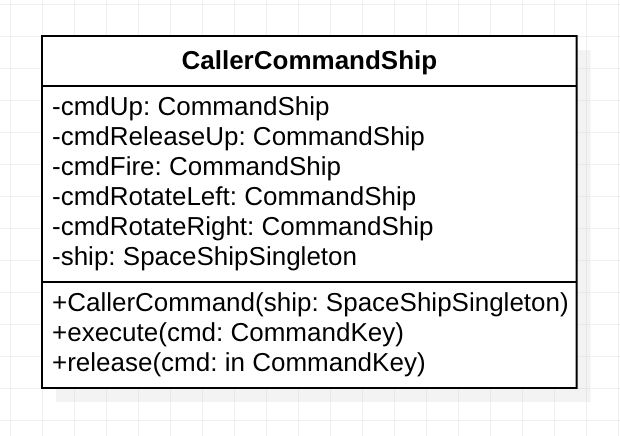
Oltre al MovementKeyListener, che aggiunge comandi a una lista di comandi relativi alla navicella vi è il CallerCommand, ovvero un telecomando che, traducendo il CommandKey, richiama il giusto insieme di azioni specifiche per quel comando.

Tramite tastiera possono essere eseguiti 5 principali comandi:

-L’accelerazione della navicella

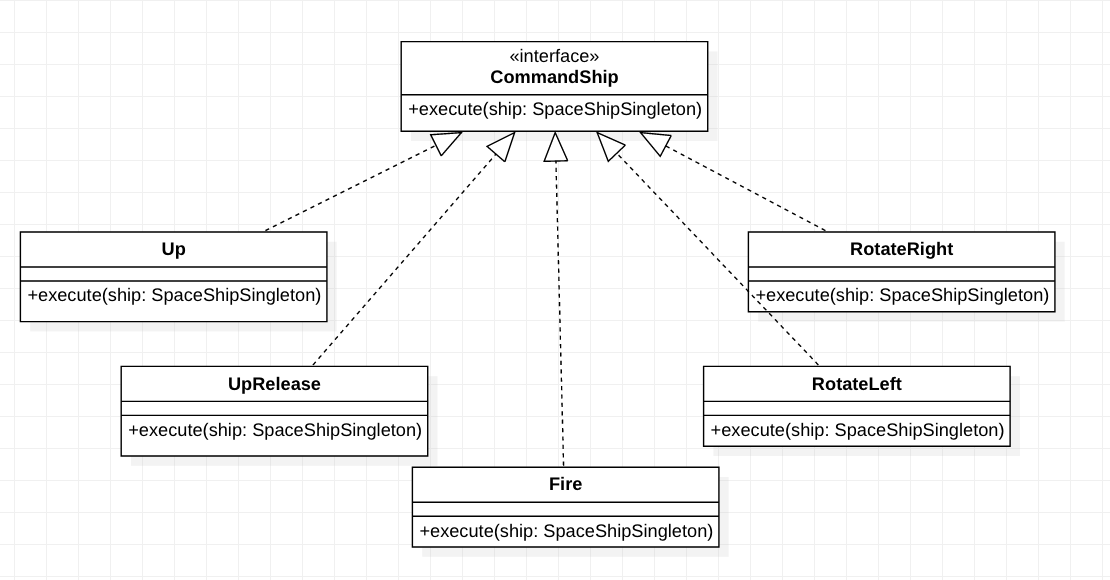
-La decelerazione della navicella

-La rotazione a sinistra

-La rotazione a destra

-Lo sparo

Quest’ultimo è stato composto da 5 campi CommandShip ed ognuno eseguirà la propria implementazione.



Questi 5 comandi principali possono essere richiamati tramite le frecce direzionali destra o sinistra per le rotazioni oppure con quella direzionata verso l’alto per l’accelerazione.

Inoltre vi è la possibilità di usare le lettere A (rotate left), W (acelerate), D (rotate right), K (shoot).

Design specifico del suono

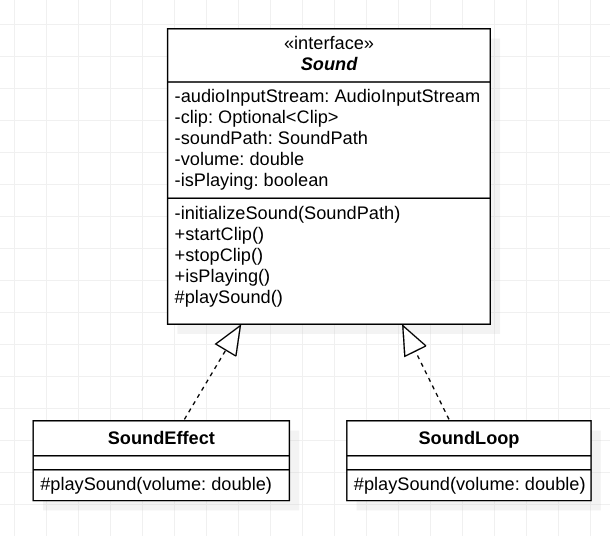
In questo software il suono è presente sin dall’avvio dello stesso.

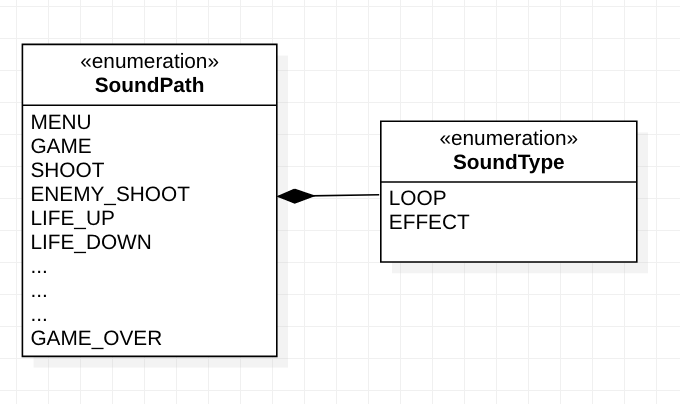
Il pattern usato è stato sempre il Command per poter accendere o spegnere il relativo suono, a prescindere dal tipo di quest’ultimo.

Ho creato un’ interfaccia astratta Sound composta dai metodi di base del suono, come lo start, la clip o il volume dello stesso.

Dopodiché ho implementato due classi, SoundEffect e SoundLoop.

Come si può evincere dal nome le due classi eseguiranno il suono una sola volta o in loop.

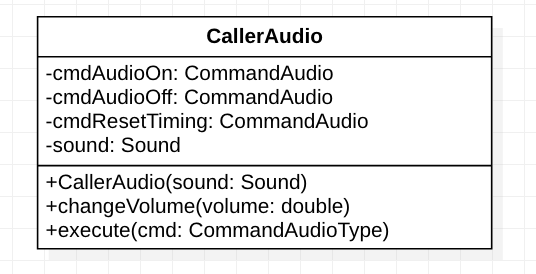


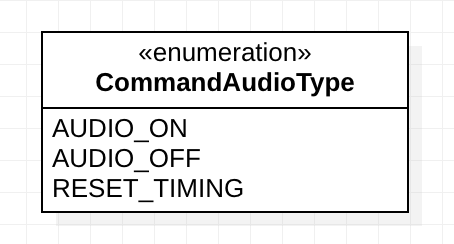


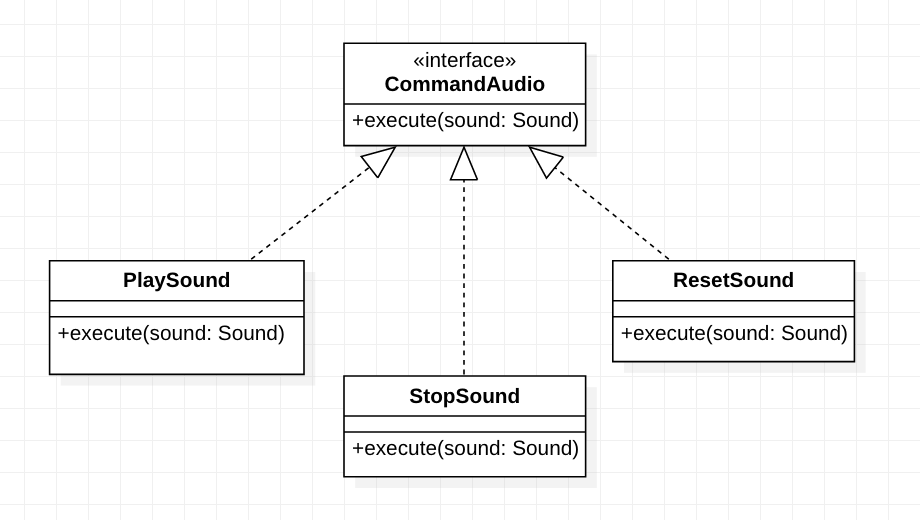
SoundLoop è la classe di appartenenza del suono durante il gioco o quello del menu di pausa mentre SoundEffect è la classe di appartenenza di tutti gli effetti di gioco come sparo, raccolta di un perk, ecc.

L’obiettivo posto era rendere indipendenti i vari suoni evitando che, alla partenza di un nuovo effetto, quello prima si stoppasse.

Ho evitato questo problema componendo un CallerAudio per ogni effetto, ed eseguendo un comando ON ogni volta che avveniva un certo evento come collisione con perk o con nemici.







Design delle collisioni

Per effettuare il controllo sulle collisioni tra gli oggetti di gioco sono partito creando una classe CollisionChecker nella quale ho implementato 2 metodi per controllare tutte le possibili collisioni tra gli oggetti.

Ho basato il collision cheker su RectBoundingBox e CircleBoudingBox, ovvero i contorni relativi a ogni tipo di oggetto di gioco.

I metodi confrontano anche figure sottoposte a rotazione, quindi trasfomate.

Avendo strutturato l’interfaccia WorldEvent. Tutte le classi che implementano quest’ultima sono considerati eventi e possono essere usati per mettere in comunicazione i vari oggetti di gioco.

Ad esempio

Metodologia di lavoro

Il lavoro è stato suddiviso come elencato

3.2 Metodologia di lavoro

Ci aspettiamo, leggendo questa sezione, di trovare conferma alla divisione operata nella sezione del design di dettaglio, e di capire come `e stato svolto il lavoro di integrazione. Andr`a realizzata una sotto-sezione separata per ciascuno studente che identifichi le porzioni di progetto sviluppate, separando quelle svolte in autonomia da quelle sviluppate in collaborazio- ne. Diversamente dalla sezione di design, in questa `e consentito elencare package/classi, se lo studente ritiene sia il modo piu` efficace di convogliare l’informazione. Si ricorda che l’impegno deve giustificare circa 40-50 ore di sviluppo (`e normale e fisiologico che approssimativamente la meta` del tempo sia impiegata in analisi e progettazione).

Elementi positivi

* Si identifica con precisione il ruolo di ciascuno all’interno del gruppo, os- sia su quale parte del progetto ciascuno dei componenti si `e concentrato maggiormente.
* La divisione dei compiti `e equa, ossia non vi sono membri del gruppo che hanno svolto molto piu` lavoro di altri.
* La divisione dei compiti `e coerente con quanto descritto nelle parti precedenti della relazione.   
  1Testare in modo automatico le interfacce grafiche `e possibile (si veda, come esempio, https://github.com/TestFX/TestFX), semplicemente nel corso non c’`e modo e tempo di introdurvi questo livello di complessit`a. Il fatto che non vi sia stato insegnato come farlo non implica che sia impossibile!   
  18

testata automaticamente perch ́e `e impossibile farlo .

Immagine

* La divisione dei compiti `e realistica, ossia le dipendenze fra le parti sviluppate sono minime.
* Si identifica quale parte del software `e stato sviluppato da tutti i componenti insieme.
* Si spiega in che modo si sono integrate le parti di codice sviluppa- te separatamente, evidenziando eventuali problemi. Ad esempio, una strategia `e convenire sulle interfacce da usare (ossia, occuparsi insieme di stabilire l’architettura) e quindi procedere indipendentemente allo sviluppo di parti differenti. Una possibile problematica potrebbe es- sere una dimenticanza in fase di design architetturale che ha costretto ad un cambio e a modifiche in fase di integrazione. Una situazione simile `e la norma nell’ingegneria di un sistema software non banale, ed il processo di progettazione top-down con raffinamento successivo `e il cos`ı detto processo “a spirale”.
* Si descrive in che modo `e stato impiegato il DVCS.   
  Elementi negativi
* Non si chiarisce chi ha fatto cosa.
* C’`e discrepanza fra questa sezione e le sezioni che descrivono il design   
  dettagliato.
* Tutto il progetto `e stato svolto lavorando insieme invece che assegnando una parte a ciascuno.
* Non viene descritta la metodologia di integrazione delle parti sviluppate indipendentemente.
* Uso superficiale del DVCS.

3.3 Note di sviluppo

Questa sezione, come quella riguardante il design dettagliato va svolta sin- golarmente da ogni membro del gruppo.

Ciascuno dovra` mettere in evidenza eventuali particolarita` del suo metodo di sviluppo, ed in particolare:

• 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:

19

* –  Progettazione con generici, ad esempio costruzione di nuovi tipi generici, e uso di generici bounded. Uso di classi generiche di libreria non `e considerato avanzato.
* –  Uso di lambda expressions
* –  Uso di Stream, di Optional o di altri costrutti funzionali
* –  Uso della reflection
* –  Definizione ed uso di nuove annotazioni
* –  Uso del Java Platform Module System
* –  Uso di parti di libreria non spiegate a lezione (networking, com- pressione, parsing XML, eccetera...)
* –  Uso di librerie di terze parti (incluso JavaFX): Google Guava, Apache Commons...
* –  Uso di build systems

Si faccia molta attenzione a non scrivere banalita`, elencando qui featu- res di tipo “core”, come le eccezioni, le enumerazioni, o le inner class: nessuna di queste `e considerata avanzata.

* Descrivere molto brevemente le librerie utilizzate nella propria parte di progetto, se non trattate a lezione (ossia, se librerie di terze parti e/o se componenti del JDK non visti, come le socket). Si ricorda che l’utilizzo di librerie `e valutato positivamente.
* Sviluppo di algoritmi particolarmente interessanti non forniti da alcuna libreria (spesso pu`o convenirvi chiedere sul forum se ci sia una libreria per fare una certa cosa, prima di gettarvi a capofitto per scriverla voi stessi).   
  In questa sezione, dopo l’elenco, `e anche bene evidenziare eventuali pezzi di codice “riadattati” (o scopiazzati...) da Internet o da altri progetti, pratica che tolleriamo ma che non raccomandiamo. I pattern di design, invece non vanno messi qui. L’uso di pattern di design (come suggerisce il nome) `e 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   
  20
* Si descrivono aspetti particolarmente complicati o rilevanti relativi al- l’implementazione, ad esempio, in un’applicazione performance criti- cal, un uso particolarmente avanzato di meccanismi di caching, oppure l’implementazione di uno specifico algoritmo.
* Se si `e utilizzato un particolare algoritmo, se ne cita la fonte originale. Ad esempio, se si `e usato Mersenne Twister per la generazione dei numeri pseudo-random, si cita [MN98].
* Si identificano parti di codice prese da altri progetti, dal web, o comun- que scritte in forma originale da altre persone. In tal senso, si ricorda che agli ingegneri non `e richiesto di re-inventare la ruota continuamen- te: se si cita debitamente la sorgente `e tollerato fare uso di di snippet di codice per risolvere velocemente problemi non banali. Nel caso in cui si usino snippet di codice di qualit`a 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 `e largamente meglio fare uso di una libreria che copiarsi pezzi di codice: qualora vi sia scelta (e tipicamente c’`e), si preferisca la prima via.   
  Elementi negativi
* Si elencano feature core del linguaggio invece di quelle segnalate. Esem- pi 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 `e 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 `e in ogni caso inteso come sviluppato da voi.   
    21
* 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 (o sedicenti tali), ai blog dedicati allo sviluppo di soluzioni e applicazioni (inclusi blog dedicati ad Android e allo sviluppo di videogame), nonch ́e ai social network. Conseguentemente, `e molto conveniente citare una fonte ed usarla invece di tentare di spacciare per proprio il lavoro di altri.
* Si elencano design pattern

\chapter{Analisi}

In questo capitolo andrà fatta l'analisi dei requisiti e quella del problema, ossia verranno elencate le cose che l'applicazione dovrà fare (requisiti) e verrà descritto il dominio applicativo (analisi del problema).

%

In fase di analisi, è molto importante tenere a mente che non vi deve essere alcun riferimento al design né tantomeno alle tecnologie implementative, ovvero, non si deve indicare come il software sarà internamente realizzato.

%

La fase di analisi, infatti, \textit{precede} qualunque azione di design o di implementazione.

\section{Requisiti}

Nell'analisi dei \emph{requisiti} dell'applicazione si dovrà spiegare cosa l'applicazione dovrà fare.

%

Non ci si deve concentrare sui particolari problemi, ma esclusivamente su cosa si desidera che l'applicazione faccia.

%

È consigliato descrivere separatamente i requisiti funzionali (quelli che descrivono l'effettivo

comportamento dell'applicazione) da quelli non funzionali (requisiti che non riguardano direttamente

aspetti comportamentali, come sicurezza, performance, eccetera).

\subsection\*{Elementi positivi}

\begin{itemize}

\item Si fornisce una descrizione in linguaggio naturale di ciò che il software dovrà fare.

\item Gli obiettivi sono spiegati con chiarezza, per punti.

\item Se il software è stato commissionato o è destinato ad un utente o compagnia specifici, il committente viene nominato.

\item Se vi sono termini il cui significato non è immediatamente intuibile, essi vengono spiegati.

\item Vengono descritti separatamente requisiti funzionali e non funzionali.

\item Considerato a un paio di pagine un limite ragionevole alla lunghezza della parte sui requisiti, in quello spazio si deve cercare di chiarire \textit{tutti} gli aspetti dell'applicazione, non lasciando decisioni che impattano la parte ``esterna'' alla discussione del design (che dovrebbe solo occuparsi della parte ``interna'').

\end{itemize}

\subsection\*{Elementi negativi}

\begin{itemize}

\item Si forniscono indicazioni circa le soluzioni che si vogliono adottare

\item Si forniscono dettagli di tipo tecnico o implementativo (parlando di classi, linguaggi di programmazione, librerie, eccetera)

\end{itemize}

\subsection\*{Esempio}

Il software, commissionato dal gestore del centro di ricerca ``Aperture Laboratories Inc.''\footnote{\url{http://aperturescience.com/}}, mira alla costruzione di una intelligenza artificiale di nome GLaDOS (Genetic Lifeform and Disk Operating System).

%

Per intelligenza artificiale si intende un software in grado di assumere decisioni complesse in maniera semi autonoma sugli argomenti di sua competenza, a partire dai vincoli e dagli obiettivi datigli dall'utente.

\subsubsection{Requisiti funzionali}

\begin{itemize}

\item La suddetta intelligenza artificiale dovrà occuparsi di coordinare le attività all'interno

delle camere di test di Aperture, guidando l'utente attraverso un certo numero di sfide di

difficoltà crescente. Una camera di test è un ambiente realizzato da Aperture Laboratories Inc. al

fine di mettere alla prova le proprie tecnologie di manipolazione dell'ambiente. All'interno della

camera di test, un soggetto qualificato è incaricato di sfruttare gli strumenti messi a

disposizione da Aperture per risolvere alcuni rompicapi. I rompicapi sono di tipo fisico (ad

esempio, manipolazione di oggetti, pressione di pulsanti, azionamento di leve), e si ritengono

conclusi una volta che il soggetto riesce a trovare l'uscita dalla camera di test.

\item Il piano preciso ed il numero delle sfide sarà variabile, e GLaDOS dovrà essere in grado di adattarsi dinamicamente e di fornire indicazioni di guida.

\item La personalità di GLaDOS dovrà essere modificabile.

\item GLaDOS dovrà essere in grado di comunicare col reparto cucina di Aperture, per ordinare torte da donare agli utenti che completassero l'ultima camera di test con successo.

\end{itemize}

\subsubsection{Requisiti non funzionali}

\begin{itemize}

\item GLaDOS dovrà essere estremamente efficiente nell'uso delle risorse. Le specifiche tecniche parlano della possibilità di funzionare su dispositivi alimentati da una batteria a patata.

\end{itemize}

\section{Analisi e modello del dominio}

In questa sezione si descrive il modello del \textit{dominio

applicativo}, descrivendo le \textit{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 \textbf{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 \textit{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 \textbf{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.

\subsection\*{Elementi positivi}

\begin{itemize}

\item Viene descritto accuratamente il modello del dominio.

\item Alcuni problemi, se non risolubili in assoluto o nel monte ore, vengono dichiarati come problemi che non saranno risolti o sarano risolti in futuro.

\item Si modella il dominio in forma di UML, descrivendolo appropriatamente.

\end{itemize}

\subsection\*{Elementi negativi}

\begin{itemize}

\item Manca una descrizione a parole del modello del dominio.

\item Manca una descrizione UML delle entità del dominio e delle relazioni che intercorrono fra loro.

\item Vengono elencate soluzioni ai problemi, invece della descrizione degli stessi.

\item Vengono presentati elementi di design, o peggio aspetti implementativi.

\item 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, o cose che non siano equivalenti ad interfacce).

\end{itemize}

\subsection\*{Esempio}

GLaDOS dovrà essere in grado di accedere ad un'insieme di camere di test.

%

Tale insieme di camere prende il nome di percorso.

%

Ciascuna camera è composta di challenge successivi.

%

GLaDOS è responsabile di associare a ciascun challenge un insieme di consigli (suggestions) destinati all'utente (subject), dipendenti da possibili eventi.

%

GLaDOS dovrà poter comunicare coi locali cucina per approntare le torte.

%

Le torte potranno essere dolci, oppure semplici promesse di dolci che verranno disattese.

Gli elementi costitutivi il problema sono sintetizzati in \Cref{img:analysis}.

La difficoltà primaria sarà quella di riuscire a correlare lo stato corrente dell'utente e gli eventi in modo tale da generare i corretti suggerimenti.

%

Questo richiederà di mettere in campo appropriate strategie di intelligenza artificiale.

Data la complessità di elaborare consigli via AI senza intervento umano, la prima versione del software fornita prevederà una serie di consigli forniti dall'utente.

Il requisito non funzionale riguardante il consumo energetico richiederà studi specifici sulle performance di GLaDOS che non potranno essere effettuati all'interno del monte ore previsto: tale feature sarà oggetto di futuri lavori.

\begin{figure}[h]

\centering{}

\includegraphics{img/analysis.pdf}

\caption{Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro}

\label{img:analysis}

\end{figure}d