

Meta-relazione per
“Programmazione ad Oggetti”

Giovanni Di Santi

21 settembre 2020

Sommario

Questo documento è una relazione di meta livello, ossia una relazione che spiega come scrivere la relazione. Lo scopo di questo documento è quello di aiutare gli studenti a comprendere quali punti trattare nella loro relazione, ed in che modo farlo, evitando di perdere del tempo prezioso in prolisse discussioni di aspetti marginali tralasciando invece aspetti di maggior rilievo. Per ciascuna delle sezioni del documento sarà fornita una descrizione di ciò che ci si aspetta venga prodotto dal team di sviluppo, assieme ad un elenco (per forza di cose non esaustivo) di elementi che *non* dovrebbero essere inclusi.

Il modello della relazione segue il processo tradizionale di ingegneria del software fase per fase (in maniera ovviamente semplificata). La struttura della relazione non è indicativa ma *obbligatoria*. Gli studenti dovranno produrre un documento che abbia la medesima struttura, non saranno accettati progetti la cui relazione non risponda al requisito suddetto. Lo studente attento dovrebbe sforzarsi di seguire le tappe suggerite in questa relazione anche per l'effettivo sviluppo del progetto: oltre ad una considerevole semplificazione del processo di redazione di questo documento, infatti, il gruppo beneficerà di un processo di sviluppo più solido e collaudato, di tipo top-down.

La meta-relazione verrà fornita corredata di un template \LaTeX per coloro che volessero cimentarsi nell'uso. L'uso di \LaTeX è vantaggioso per chi ama l'approccio “what you mean is what you get”, ossia voglia disaccoppiare il contenuto dall'effettivo rendering del documento, accollando al motore \LaTeX l'onere di produrre un documento gradevole con la struttura ed il contenuto forniti. Chi non volesse installare l'ambiente di compilazione in locale può valutare l'utilizzo dell'applicazione web Overleaf. L'eventuale utilizzo di \LaTeX non è fra i requisiti, non è parte del corso di Programmazione ad Oggetti, e non sarà ovviamente valutato. I docenti accetteranno qualunque relazione in formato standard Portable Document Format (pdf), indipendentemente dal software con cui tale documento sarà redatto.

Indice

1	Analisi	2
1.1	Requisiti	2
1.2	Analisi e modello del dominio	2
2	Design	5
2.1	Architettura	5
2.2	Design dettagliato	7
3	Sviluppo	11
3.1	Testing automatizzato	11
3.2	Metodologia di lavoro	12
3.3	Note di sviluppo	13
4	Commenti finali	17
4.1	Autovalutazione e lavori futuri	17
4.2	Difficoltà incontrate e commenti per i docenti	17
A	Guida utente	19

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare un keepass per salvare in modo sicuro le password. Al momento i keepass desktop più usati sono keepass2 e keepassxc.

Requisiti funzionali

- Gestione account con possibilità di suddividere gli account in gruppi.
- Funzione per generare password e nickname in maniera casuale.
- Import ed export del database in XML e in un formato simile a KDBX.
- Controllo robustezza password al momento dell'inserimento
- Salvataggio dei dati in un database locale criptato con algoritmi a scelta
- Sezione Statistics che mostra le statistiche relative al proprio database (Es. il numero di account salvati)

Requisiti non funzionali

- bho

1.2 Analisi e 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, dovrete 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

- Viene descritto accuratamente il modello del dominio.
- Alcuni problemi, se non risolvibili in assoluto o nel monte ore, vengono dichiarati come problemi che non saranno risolti o saranno risolti in futuro.
- Si modella il dominio in forma di UML, descrivendolo appropriatamente.

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 elencate soluzioni ai problemi, invece della descrizione degli stessi.
- Vengono presentati elementi di design, o peggio aspetti implementativi.
- Viene mostrato uno schema UML che include elementi implementativi o non utili alla descrizione del dominio, ma volti alla soluzione (non

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

devono vedersi, ad esempio, campi o metodi privati, o cose che non siano equivalenti ad interfacce).

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 Figura 1.1.

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.

Capitolo 2

Design

In questo capitolo si spiegano le strategie messe in campo per realizzare la soluzione ai problemi identificati nell'analisi.

Si parte da una visione architetturale, il cui scopo è informare il lettore di quale sia il funzionamento dell'applicativo realizzato ad alto livello. In particolare, è necessario descrivere accuratamente in che modo i componenti principali del sistema si coordinano fra loro. A seguire, si dettagliano alcune parti del design, quelle maggiormente rilevanti al fine di chiarificare la logica con cui sono stati risolti i problemi dell'applicazione.

2.1 Architettura

Per la realizzazione di KeePassJ abbiamo scelto di utilizzare il pattern architetturale Model-View-Controller (MVC).

Elementi positivi

- Si mostrano pochi, mirati schemi UML dai quali si deduce con chiarezza quali sono le parti principali del software e come interagiscono fra loro.
- Si mette in evidenza se e come il pattern architetturale model-view-controller è stato applicato, anche con l'uso di un UML che mostri le interfacce principali ed i rapporti fra loro.
- Si discute se sia semplice o meno, con l'architettura scelta, sostituire in blocco la view: in un MVC ben fatto, controller e modello non dovrebbero in alcun modo cambiare se si transitasse da una libreria grafica ad un'altra (ad esempio, da Swing a JavaFX, o viceversa).

Elementi negativi

- L'architettura è fatta in modo che sia impossibile riusare il modello per un software diverso che affronta lo stesso problema.
- L'architettura è tale che l'aggiunta di una funzionalità sul controller impatta pesantemente su view e/o modello.
- L'architettura è tale che la sostituzione in blocco della view impatta sul controller o, peggio ancora, sul modello.
- Si presentano UML caotici, difficili da leggere.
- Si presentano UML in cui sono mostrati elementi di dettaglio non appartenenti all'architettura, ad esempio includenti campi o con metodi che non interessano la parte di interazione fra le componenti principali del software.
- Si presentano schemi UML con classi (nel senso UML del termine) che “galleggiano” nello schema, non connesse, ossia senza relazioni con il resto degli elementi inseriti.
- Si presentano elementi di design di dettaglio, ad esempio tutte le classi e interfacce del modello o della view.
- Si discutono aspetti implementativi, ad esempio eventuali librerie usate oppure dettagli di codice.

Esempio

L'architettura di GLaDOS segue il pattern architetturale MVC. Più nello specifico, a livello architetturale, si è scelto di utilizzare MVC in forma “ECB”, ossia “entity-control-boundary”¹. GLaDOS implementa l'interfaccia AI, ed è il controller del sistema. Essendo una intelligenza artificiale, è una classe attiva. GLaDOS accetta la registrazione di Input ed Output, che fanno parte della “view” di MVC, e sono il “boundary” di ECB. Gli Input rappresentano delle nuove informazioni che vengono fornite all'IA, ad esempio delle modifiche nel valore di un sensore, oppure un comando da parte dell'operatore. Questi input infatti forniscono eventi. Ottenere un evento è un'operazione bloccante: chi la esegue resta in attesa di un effettivo evento. Di fatto, quindi, GLaDOS si configura come entità *reattiva*. Ogni volta che c'è un

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

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

cambio alla situazione del soggetto, GLaDOS notifica i suoi Output, informandoli su quale sia la situazione corrente. Conseguentemente, GLaDOS è un “observable” per Output.

Con questa architettura, possono essere aggiunti un numero arbitrario di input ed output all'intelligenza artificiale. Ovviamente, mentre l'aggiunta di output è semplice e non richiede alcuna modifica all'IA, la presenza di nuovi tipi di evento richiede invece in potenza aggiunte o rifiniture a GLaDOS. Questo è dovuto al fatto che nuovi Input rappresentano di fatto nuovi elementi della business logic, la cui alterazione od espansione inevitabilmente impatta il controller del progetto.

In Figura 2.1 è esemplificato il diagramma UML architetturale.

2.2 Design dettagliato

Giovanni Di Santi

Il mio compito principale del progetto è stato quello di gestire la parte crittografica e definire la struttura dell'header del database.

CryptoCipher

CryptoCipher è l'interfaccia che descrive i metodi necessari per effettuare l'encryption e la decryption di un array di byte.

Ogni implementazione disponibile di questa interfaccia è un AEAD Cipher (Authenticated Encryption with Associated Data).

Ho scelto questo schema di encryption per rendere il database resistente ad attacchi del tipo **CCA** (Chosen Ciphertext Attack), cifrando il contenuto del database e autenticando sia il contenuto che l'header (Associated Data). Attualmente i cipher disponibili sono:

- ChaCha20-Poly1305.
- AES-GCM.
- AES-256-CBC-HMAC-SHA-512.

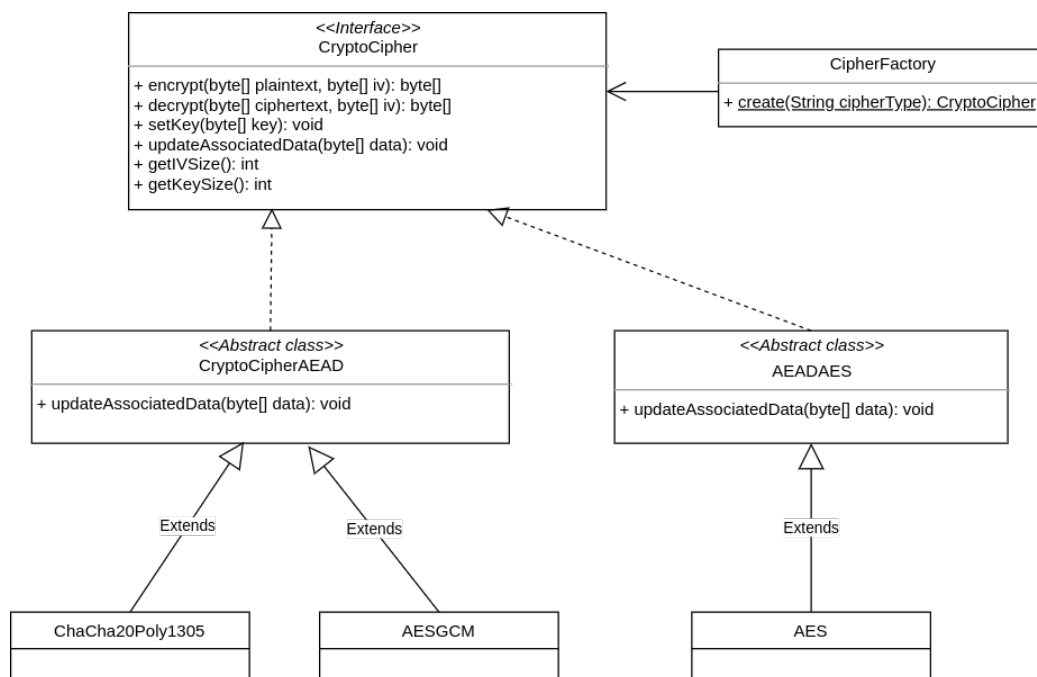


Figura 2.2: rappresentazione UML del pattern factory per creare un `CryptoCipher`

Esistono due abstract class diverse per implementare un **CryptoCipher**, poiché la costruzione di **AES** che sarebbe AES-256-CBC-HMAC-SHA-512 è manuale, mentre **ChaCha20-Poly1305** e **AES-GCM** sono implementate direttamente in `openjdk11`. La classe astratta **AEADAES**, permette di essere estesa per costruire altri cryptosystem come AES-192-CBC-HMAC-SHA-384, tuttavia ho deciso di estendere solo lo schema più sicuro.

Nonostante i dati da cifrare e decifrare sono nella pratica degli `{Input,Output}Stream`, non ho usato le classi `CipherOutputStream` e `CipherInputStream` per:

- Rendere più semplice il suo utilizzo.
- Facilitare il testing delle varie implementazioni.

KDF

KDF.

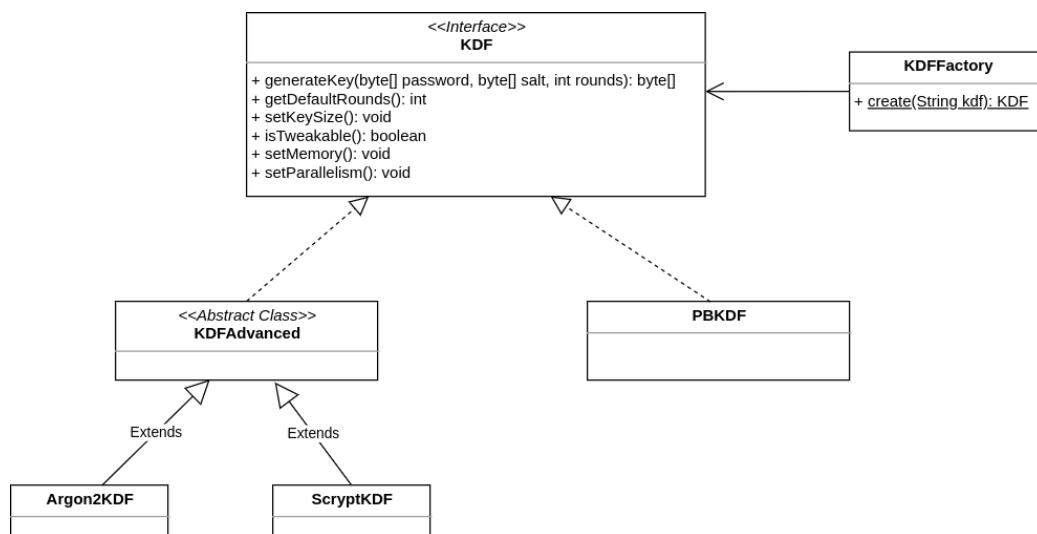


Figura 2.3: rappresentazione UML del pattern factory per creare un KDF

Contro-esempio: pessimo diagramma UML

In Figura 2.5 è mostrato il modo **sbagliato** di fare le cose. Questo schema è fatto male perché:

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

Figura 2.4: 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.
- Se sono stati eseguiti test manuali di rilievo, si elencano descrivendo brevemente la ragione per cui non sono stati automatizzati. Ad esempio, se tutto il team sviluppa e testa su uno stesso sistema operativo e si sono svolti test manuali per verificare, ad esempio, il corretto funzionamento dell'interfaccia grafica o di librerie native su altri sistemi operativi, può avere senso menzionare la cosa.

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* farlo¹.
- 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 Metodologia di lavoro

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

Elementi positivi

- Si identifica con precisione il ruolo di ciascuno all'interno del gruppo, ossia su quale parte del progetto ciascuno dei componenti si è concentrato maggiormente.
- La divisione dei compiti è equa, ossia non vi sono membri del gruppo che hanno svolto molto più lavoro di altri.
- La divisione dei compiti è coerente con quanto descritto nelle parti precedenti della relazione.

¹Testare 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 introdurre questo livello di complessità. Il fatto che non vi sia stato insegnato come farlo non implica che sia impossibile!

- La divisione dei compiti è realistica, ossia le dipendenze fra le parti sviluppate sono minime.
- Si identifica quale parte del software è stato sviluppato da tutti i componenti insieme.
- Si spiega in che modo si sono integrate le parti di codice sviluppate separatamente, evidenziando eventuali problemi. Ad esempio, una strategia è 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 essere una dimenticanza in fase di design architetturale che ha costretto ad un cambio e a modifiche in fase di integrazione. Una situazione simile è la norma nell'ingegneria di un sistema software non banale, ed il processo di progettazione top-down con raffinamento successivo è il così detto processo "a spirale".
- Si descrive in che modo è stato impiegato il DVCS.

Elementi negativi

- Non si chiarisce chi ha fatto cosa.
- C'è discrepanza fra questa sezione e le sezioni che descrivono il design dettagliato.
- Tutto il progetto è 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 **singolarmente da ogni membro del gruppo**.

Ciascuno dovrà mettere in evidenza eventuali particolarità del suo metodo di sviluppo, ed in particolare:

- **Elencare** 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. 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 della reflection
- Definizione ed uso di nuove annotazioni
- Uso del Java Platform Module System
- Uso di parti di libreria non spiegate a lezione (networking, compressione, 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 banalità, elencando qui features di tipo “core”, come le eccezioni, le enumerazioni, o le inner class: nessuna di queste è 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 è valutato *positivamente*.
- Sviluppo di algoritmi particolarmente interessanti *non forniti da alcuna libreria* (spesso può 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 è anche bene evidenziare eventuali pezzi di codice “riadattati” (o scopiati...) da Internet o da altri progetti, pratica che tolleriamo ma che non raccomandiamo.

Elementi positivi

- Si elencano gli aspetti avanzati di linguaggio che sono stati impiegati
- Si elencano le librerie che sono state utilizzate
- Si descrivono aspetti particolarmente complicati o rilevanti relativi all'implementazione, ad esempio, in un'applicazione performance critical, un uso particolarmente avanzato di meccanismi di caching, oppure l'implementazione di uno specifico algoritmo.

- Se si è utilizzato un particolare algoritmo, se ne cita la fonte originale. Ad esempio, se si è usato Mersenne Twister per la generazione dei 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 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 (o sedicenti tali), ai blog dedicati allo sviluppo di soluzioni e applicazioni (inclusi blog dedicati ad Android e allo sviluppo di videogame), nonché ai social network. Conseguentemente, è *molto* conveniente *citare* una fonte ed usarla invece di tentare di spacciare per proprio il lavoro di altri.

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 omissis. 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.