Relazione progetto di Programmazione Avanzata 2022/2023

Francesco Corrini

Abstract

Il progetto consiste in un insieme di tre software scritti nei linguaggi Java, C++ ed Haskell. Il primo è un piccolo videogioco implementato in Java nel quale si mostrano i concetti di ereditarietà e sottotipazione mediante oggetti, mostri e personaggi. Inoltre si utilizzeranno i costrutti messi a disposizione dal linguaggio. In C++ è stato scritto un software che implementa analisi, identificazione e simulazione di processi stocastici AR, MA, ARMA e ARMAX studiati nel corso di IMAD. Questo software mette in evidenza i concetti di ereditarietà multipla, pubblica e privata oltre all'uso dei template. In Haskell è stato implementatato il calcolo dell'equazione alle differenze di Riccati: questa, molto utilizzata nell'ambito del controllo ottimo (cfr Controlli Automatici) permette di calcolare la matrice di covarianza dello stato di un sistema dinamico (la quale è variabile nel tempo) in modo ricorsivo utilizzando il valore della matrice all'istante precedente.

1 Java - un piccolo videogame

Il software implementato consiste in un videogioco nel quale il giocatore interpreta un personaggio che può essere un guerriero o un mago: questo deve attraversare diversi luoghi (nel codice un luogo è definito Stanza), nelle quali affronterà diversi tipi di mostri. Lo scopo del gioco è sconfiggere il boss di ogni stanza per proseguire nella successiva ed alla fine affrontare il boss finale. Nei successivi paragrafi di questa sezione saranno descritte le varie componenti di questo software, ognuna delle quali implementa alcuni dei concetti, costrutti e pattern studiati durante il corso.

Mostri e boss Un mostro è definito dall'interfaccia MostroInterface che definisce i metodi necessari perché una classe si comporti come un mostro. La classe Mostro implementa quest'interfaccia e ne è quindi un sottotipo. All'interno della classe Mostro è presente un record (chiamato DatiMostro) che specifica i campi del mostro (vita, attacco, esperienza, ...). L'interfaccia mette a disposizione i metodi per accedere a questi campi e soprattutto i metodi che permettono al mostro di eseguire un attacco (per danneggiare il giocatore) e per ricevere danno dal giocatore (sulla base dell'attacco del giocatore vengono tolti alcuni punti vita). Alcuni mostri, se sconfitti, lasciano al giocatore un oggetto tramite la funzione getDrop la quale restituisce l'oggetto se il mostro ne ha uno mentre lancia

un'eccezione SenzaOggettoException (gestita dal chiamante) se non lo ha. Per creare un mostro (quando la battaglia è avviata) si è utilizzato il Factory Pattern: si ha infatti la classe FactoryMostro (implementata come singleton cosicché ne esista una sola istanza) che fornisce il metodo getMostro il quale (tramite generazione di numeri casuali) restituisce un mostro casuale. Poiché i mostri variano in base alla stanza in FactoryMostro si hanno 3 metodi getMostro differenti (facendo overload del metodo) ognuno dei quali prende come argomento una stanza differente così da creare mostri diversi in base alla stanza.
E' stata inoltre implementata l'interfaccia BossInterface la quale estende l'interfaccia MostroInterface. Questo comporta che ogni classe che implementa BossInterface sarà anche sottotipo di MostroInterface. In questa interfaccia non è stato implementato nulla, tuttavia può essere utilizzata per implementare delle features (ad esempio attacchi speciali) proprie dei soli boss e non dei mostri. Quattro classi implementano questa interfaccia e corrispondono ai quattro boss del gioco.

Personaggi, oggetti e armi Ad affrontare i mostri è il personaggio le cui azioni sono controllate dal giocatore. Ciò che un personaggio può fare è definito nell'interfaccia PersonaggioInterface (attaccare e ricevere colpi, equipaggiare armi, ricevere denaro, esperienza o oggetti, ...). Ad implementare questi metodi è la classe astratta Personaggio. La classe astratta è quindi estesa dalle classi Guerriero e Mago che ereditano il codice di Personaggio (diventando allo stesso tempo dei sottotipi). Le due classi si differenziano unicamente per il metodo equipaggiaArma (che subisce override): il Guerriero potrà equipaggiare spade mentre il mago avrà a disposizione i bracciali.
A Il Personaggio ha tra i sui campi una lista (implementata mediante ArrayList di JCF) che rappresenta l'inventario che contiene gli oggetti del personaggio: questi possono essere raccolti dai mostri o acquistati in un negozio. Una classe è un oggetto (quindi un sottotipo) se implementa l'interfaccia OggettoInterface. Ad implementare quest'interfaccia è la classe astratta Oggetto che definisce i campi (nome, costo, ...) dell'oggetto ed i metodi principali. OggettoInterface estende anche (e quindi diventa sottotipo) l'interfaccia CompareTo definendo un metodo utile a comparare (per poi eseguire un ordinamento tramite il metodo Collections.sort) gli oggetti tra di loro. E' implementata poi una classe astratta Cura che estende Oggetto ed è estesa da Pozione e SuperPozione: Cura esegue l'override del metodo usa implementando la funzione dell'oggetto (far recuperare vita al Personaggio), Pozione e SuperPozione riutilizzano lo stesso codice di Cura andando a modificare il numero di punti vita recuperati (senza ereditarietà avrei dovuto duplicare il codice di Cura in entrambe le sottoclassi).
Particolari oggetti sono le armi le quali sono identificate dall'interfaccia ArmaInterface che estende OggettoInterface. Un'arma è quindi un oggetto alla quale si aggiunge il metodo getAttacco che aggiunge danno agli attacchi del Personaggio quando equipaggiata. Quest'interfaccia è implementata dalle classi astratte Spada e Bracciale (le quali estendono

anche oggetto per ereditare e riutilizzare il codice, facendo però l'override del metodo usa) le quali sono estese dalle rispettive implementazioni con dei costruttori che definiscono i campi dell'arma (danno, costo, nome, ...).
Particolarmente interessanti sono le funzioni gestioneInventario e equipaggiaArma. La prima stampa il contenuto dell'inventario tramite una lambda function e permette al giocatore di scegliere se utilizzare un oggetto, ordinare l'inventario secondo il compareTo (in ordine alfabetico mettendo prima gli oggetti e poi le armi) e leggere la descrizione di un oggetto, mentre equipaggiaArma tramite una lambda function seleziona dall'inventario la lista di armi equipaggiabile dal Personaggio (in base a se questi è un Guerriero o Mago) e permette al giocatore di scegliere cosa equipaggiare.

Stanze e battaglie Come descritto in precedenza una stanza è un luogo dove il giocatore incontra diversi mostri con cui deve battagliare. L'interfaccia StanzaInterface (che definisce quali operazioni deve fare una stanza) definisce i metodi per:

- Avviare una battaglia con i mostri di questa stanza;
- Muoversi dalla stanza (spostandosi nella precedente o successiva);
- Combattere con il boss della stanza e il boss finale;
- Avere informazioni sulla stanza (se ha una stanza precedente o successiva, se si può combattere il boss, se è l'ultima stanza, ...).

Ad implementare questa interfaccia è la classe astratta Stanza che definisce un metodo privato per avviare una battaglia: questo prende in input giocatore ed un mostro (o un boss) e li fa combattere. Il mostro attaccherà sempre, mentre il giocatore potrà scegliere di utilizzare l'inventario o di scappare dalla battaglia. Al termine di ogni battaglia verranno poi aggiunta esperienza al giocatore (che può aumentare di livello incrementando le sue statistiche), oltre a guadagnare denaro e oggetti. Se però il giocatore muore la partita termina immediatamente. Questo metodo privato è utilizzato dai metodi battaglia, bossBattle e finalBattle. Sono definite poi tre classi Stanza1, Stanza2, Stanza3 che ereditano il codice da Stanza eseguendo però l'override dei metodi creaMostro e creaBoss: il primo utilizza FactoryMostro per creare un nuovo mostro sulla base della stanza, il secondo invece restituisce direttamente il boss della stanza. Da notare che nell'interfaccia il metodo creaBoss è definito come BossInterface, tutta via nelle classi i metodi hanno come tipo restituito la specifica classe del Boss (esempio in Stanza3 il metodo restituisce un Grifone). Questo è possibile grazie alla covarianza dei tipi.
Il giocatore inizia la partita nella prima stanza: una volta che avrà affrontato almeno 10 mostri potrà affrontare il boss della stanza ed una volta sconfitto potrà accedere alla stanza successiva. Sconfitto il boss della terza stanza il giocatore potrà affrontare il boss finale e finire il gioco.

Gestione della partita e del negozio Il programma ha una classe MAIN (che contiene il

main) che ha come unico scopo di lanciare il metodo game della classe Partita. All'inizio di questo metodo sono inizializzate le stanze e viene chiesto al giocatore di scegliere cosa vuole essere (Mago o Guerriero). Tutte le interazioni con il giocatore sono gestite da una classe Input (definita come singleton poiché deve essere unica in quanto gestisce l'istanza System.in tramite Scanner) che permette al giocatore di inserire numeri dalla console. La classe game permette poi al giocatore di giocare (avviare battaglie, usare l'inventario o altre operazioni delle stanze) finché o non muore in battaglia o vince la partita. Interessante è il negozio dove il giocatore può acquistare oggetti e armi utilizzando soldi guadagnati in battaglia. Poiché il negozio è unico è dichiarato come singleton.

Conclusioni Il progetto implementa i vari costrutti richiesti (ereditarietà e sottotipazione, interfacce, collections, record, overriding e overlaod) ma anche concetti avanzati come i design pattern, lambda functions, e molto altro.

2 C++ - processi stocastici

L'applicazione implementata in C++ prevede delle classi per la gestione di alcuni dei più utilizzati processi stocastici: AR, MA, ARMA e ARMAX, oltre alle funzioni di trasferimento non stocastiche che saranno di seguito chiamate X. Il programma mira a mostrare le potenzialità di C++ con i concetti di ereditarietà (pubblica, privata, multipla), l'utilizzo di templates e di strutture dati della libreria STL (in particolare i vector ed i loro iteratori) e di altri costrutti (virtual, distruttore, overloading, smart pointers, ...).

Descrizione dei modelli e delle loro funzioni I modelli implementati sono tutti modelli stocastici di cui interessa calcolare media, varianza e autocovarianza. Inoltre deve essere possibile calcolare il predittore ad un passo, che sarà utilizzato per eseguire la stima dei parametri dei modelli a partire dai dati. Infine deve essere possibile, dato un modello ed (eventualmente) un segnale di ingresso, simulare l'output del modello. Ogni modello è caratterizzato da dei parametri che moltiplicano i fattori deterministici del modello e da una componente stocastica rappresentata da un rumore gaussiano con una certa media ed una certa varianza.

Implementazione dei modelli Ogni modello è rappresentato da una classe. Inoltre l'interfaccia del modello è descritta in appositi file .h. Una classe (pseudoastratta) ProcessoStocastico definisce come virtual tutti i metodi condivisi dalle sottoclassi oltre ad avere i campi protected che rappresentano media e varianza del processo. Si ha un costruttore dichiarato come lista di inizializzazione che inizializza questi parametri ed uno di default che inizializza a media 0 e varianza 1.
La classe ARMA eredità in modo pubblico il codice di ProcessoStocastico diventandone quindi anche un sottotipo. ARMA definisce tutti i metodi (calcolaMedia, calcolaVarianza, ...) necessari a svolgere le sue funzioni. Inoltre ha al suo interno i campi protected che definiscono i parametri del modello (a e c) e due puntatori double i quali rappresentano

l'ultimo valore del predittore ad un passo calcolato (necessario per calcolare ricorsivamente il valore successivo) e l'ultimo valore del white noise generato (necessario per simulare correttamente il modello). I nuovi campi sono inizializzati sempre con lista di inizializzazione tranne i puntatori i qui valori sono dichiarati nello heap tramite il costrutto new. E' quindi necessario modificare il distruttore (definito virtual in ProcessoStocastico in modo tale da essere chiamato in ogni sottoclasse alla deallocazione dell'oggetto) in modo che esegua la delete di queste variabili per non avere memory leak. Come protected si ha inoltre un metodo che serve ad eliminare la media da un insieme di dati ricevuti in input (questi dati sono passati tramite un puntatore smart ad un oggetto vector). Questo metodo è utile durante la stima dei parametri per eliminare il trend dai dati e stimare a e c a partire su un processo a media 0, trasferendo la parte costante al white noise. Per ottimizzare il modello è stata definita una classe GradientDescent la quale ha il modello ottimizza (in cinque versioni, una per tipo di modello, eseguendo l'overload). Il metodo ottimizza sfrutta l'algoritmo iterativo di ottimizzazione "Gradient Descent" che aggiorna i parametri sulla base del valore della derivata di una funzione di costo (l'errore quadratico medio definito come differenza tra il dato in input ed il suo predittore ad un passo). Questo valore rappresenta anche la varianza del white noise del modello. A ottimizza è quindi necessario dare in input i dati (ed eventualmente l'ingresso), il modello che si vuole stimare, e quante iterazioni devono essere fatte, oltre al learning rate che moltiplica la derivata della funzione di costo.
I modelli AR ed MA sono delle versioni ridotte di ARMA, ovvero utilizzano entrambe un solo parametro (AR a e MA c) ed alcuni dei metodi, conservando però la struttura delle equazioni di varianza, covarianza, media, previsione ad un passo ecc. L'ereditarietà permette di riutilizzare il codice di ARMA per definire i metodi di queste classi, tuttavia deve essere necessario limitare l'interfaccia affinchè queste non utilizzino metodi non propri del loro modello. E' stata quindi utilizzata l'ereditarietà privata per rendere non visibili i metodi di ARMA da queste due classi ridefinendo (override) solo le funzioni necessarie (chiamando i metodi della superclasse). Anche i costruttori si comportano in questo modo chiamando le liste di inizializzazione di ARMA.
Il modello ARMAX invece rappresenta un sottotipo di ARMA (poiché ne implementa tutte le funzioni estendendone l'interfaccia). ARMAX eredita quindi in modo pubblico ARMA, oltre alla classe X che rappresenta l'ingresso esogeno. ARMAX riutilizza quasi completamente tutti i metodi di ARMA eseguendo l'override solo di alcuni (per combinare i metodi di ARMA e X). Questo è ovviamente possibile solo grazie all'ereditarietà multipla di C++ non presente in altri linguaggi come Java (poiché non considerata sicura in quanto possono avvenire dei clash tra i nomi di campi e metodi uguali nelle superclassi).

Test dell'applicazione Nel main è stata definita una funzione generica (utilizzando i template) simulaModello la quale preso in input un modello (ed eventualmente un

segnale d'ingresso) simula l'output del modello utilizzando i metodi simulaModello contenuti in ognuna delle classi. La funzione restituisce uno shared pointer contenente i dati simulati. Questo shared pointer può essere condiviso in più record dell'applicazione ed il suo contenuto sarà deallocato solo quando ogni riferimento verrà eliminato dallo stack.
Per mostrare il funzionamento dell'applicazione nel main sono state fatte le seguenti operazioni:

- Si è creato un modello AR con il costruttore parametrizzato e si è simulata una sua realizzazione con la funzione template simulaModello;
- Sono stati poi dichiarati 3 modelli con il costruttore di default (un ARMA, un AR ed un MA) e ne si sono stimati i parametri utilizzando i dati simulati dal primo modello per vedere le loro performance (ovviamente, il modello AR ha stimato meglio i parametri del suo simile rispetto agli altri due);
- Si è poi dichiarato un modello ARMAX e si è simulata una sua realizzazione utilizzando come ingresso esogeno il precedente processo AR;
- Si è utilizzata questa realizzazione per eseguire una stima dei parametri tramite due modelli ARMAX e X.

Conclusioni I risultati sono positivi per il modello AR, meno per i modelli ARMAX e ARMA (la loro funzione di costo non è convessa e quindi è preferibile utilizzare altri metodi iterativi come quello di Newton). Inoltre ARMA ha troppi parametri per stimare un AR (problema dell'identificabilità strutturale, cfr IMAD). Considerazioni sui processi stocastici a parte, in questo software è stato fatto ampio uso dei costrutti appresi durante il corso (unique e shared pointers, ereditarietà, liste di inizializzazione, distruttore e costruttore, overriding, overload ecc.) utilizzandoli in un progetto interdisciplinare che rafforza le conoscenze di entrambi i campi di studio.

3 Haskell - equazione alle differenze di Riccati

La terza parte di questo progetto riguarda l'implementazione di un piccolo software in Haskell. Si è scelto di implementare una funzione che calcola ricorsivamente la matrice di covarianza dell'errore dello stato di un sistema dinamico a tempo discreto utilizzando l'equazione alle differenze di Riccati. Questa equazione si può scrivere in diverse forme (utilizzando ad esempio il guadagno di Kalman) ma il modo più comodo di rappresentarla è il seguente: $P(t+1) = FP(t)F' - FP(t)H'(HP(t)H' + V_2)^{-1}HP(t)F' + V_1$ dove:

- F è la matrice dello stato;
- H è la matrice dell'uscita;
- V1 è la matrice di covarianza dell'errore sullo stato;
- V2 è la matrice di covarianza dell'errore sull'uscita;
- P è la matrice di covarianza dell'errore di predizione dello stato al tempo t.

Inizialmente si suppone lo stato nullo e quindi $P(0) = V_1$. Si è quindi realizzata una funzione in Haskell che prese in input le matrici sopracitate calcoli i valori di P(t) con $0 \leq t \leq \tau$. La funzione è tuttavia limitata ad input (e di conseguenza output) di matrici 2x2.

Implementazione Le matrici sono state implementate come liste di liste (ogni lista rappresenta una riga, una lista di righe la matrice). Per implementare la formula sopra descritta è stato necessario implementare alcune funzioni accessorie:

- am (sm): funzione che ricorsivamente somma (sottrae) due matrici quadrate. Ad ogni passo della ricorsione viene selezionata la prima riga delle due matrici: questa viene "zippata" (si crea una lista di tuple che accoppiano un elemento di una riga con il corrispondente dell'altra riga) e si sommano (sottraggono) i valori all'interno delle tuple, producendo la riga sommata. Si chiama poi ricorsivamente sulle righe rimanenti concatenando il risultato della sottochiamata con quello appena ottenuto.

- mm: implementa la moltiplicazione fra matrici in modo ricorsivo. Questa funzione chiama una funzione mmt con argomenti la prima matrice e la trasposta della seconda. Questa chiama una funzione cr (calcola riga) dando come argomenti la prima riga della prima matrice e l'intera seconda e concatena il risultato alla chiamata di se stessa con le righe rimanenti della prima matrice. cr a sua volta chiama una funzione cc (calcola cella) dando come argomenti la riga ricevuta in input e la prima riga della seconda matrice (che rappresenta la prima colonna della matrice originale non trasposta). Quindi cc esegue il prodotto scalare delle due liste (zippandole, moltiplicando i valori nelle tuple e sommando i valori ottenuti).

- tsp: calcola la trasposta di una matrice 2x2 invertendo gli elementi sulla diagonale secondaria.

- det: calcola il determinante di una matrice 2x2 come ad - bc.

- inv: calcola l'inversa di una matrice 2x2, invertendo gli elementi sulla diagonale principale, cambiando segno a quelli sulla diagonale secondaria e dividendo ogni elemento per il determinante.

Grazie a queste funzioni è stato possibile implementare il calcolo ricorsivo dell'equazione di Riccati. Ulteriori sviluppi possono essere fatti generalizzando le funzioni tsp, inv e det.

Conclusione

Lo svolgimento di questo progetto è stato utile per mettere in pratica gli argomenti teorici appresi durante il corso completando così la conoscenza dei metodi di programmazione avanzata in due dei linguaggi ad oggetti più utilizzati al mondo (Java e C++) e sperimentando un nuovo paradigma di programmazione differente (programmazione funzionale) per conoscerne i pregi ed i difetti. I risultati di questo progetto sono ampiamente soddisfacenti.