KALK: Progetto di Programmazione ad Oggetti

a.a. 2017/2018

Francesco Sacchetto

1136168

Nicola Cisternino

1123075

Relazione di Francesco Sacchetto.

1. Premessa

Vengono consegnate due cartelle, una chiamata "ProgettoCpp" con i files in linguaggio C++ e l'altra chiamata "ProgettoJava" con i relativi files in linguaggio Java. Nella cartella "ProgettoCpp" viene inoltre fornito un file "progetto.pro" necessario alla generazione automatica tramite `qmake` del Makefile. Il progetto è stato sviluppato per la maggior parte in laboratorio con sistema operativo Linux, compilatore GNU g++ (5.4.0), libreria Qt (5.5.1) e macchina virtuale Java (1.8.0). La restante parte del lavoro è stata eseguita da casa, da computer personali, con sistema operativo Windows 10, libreria Qt (5.9.2) e compilatore GNU g++ (5.3.0).

2. Abstract

Questo progetto rappresenta una "calcolatrice" con varie operazioni legate al tempo e alla sua gestione. L'utente ha la possibilita' di visualizzare, modificare e gestire qualsiasi tipo di orario, rappresentato tramite un orologio digitale. Queste stesse operazioni vengono offerte anche per gestire e pianificare quantita' di tempo che vanno oltre la singola giornata, inserendo una data completa su cui si possono fare interrogazioni specifiche di vari tipi. Vi sono inoltre due aggiunte: la prima e' rappresentata dalla possibilita' di inserire, gestire e controllare il fuso orario di dove si e' e dove si vorrebbe andare; l'altra si occupa del calendario lunare, fornendo all'utente tutte le informazioni necessarie per conoscere i vari aspetti delle fasi della luna.

3. Descrizione Gerarchia

La gerarchia utilizzata in questo progetto rappresenta, come spiegato nell'Abstract, varie unita' di tempo con diverse aggiunte.

- La classe base e' **Orario** i cui oggetti rappresentano un orario digitale nel formato *hh:mm:ss .* Ogni **Orario** e' caratterizzato da un numero intero che rappresenta i secondi. Per quanto riguarda i metodi piu' semplici, ve ne sono di interrogazione (*getOre, getMin* e *getSec*), di aggiornamento del campo *sec (aggiornaSec)*, di stampa (operator<<), di confronto *(operator*>) e di assegnazione *(operator=).* Vengono definiti vari costruttori, tra cui un costruttore di copia. Vi e' un metodo statico (*ConvertiInOreDate)* che serve a ottenere quante ore (ad esempio di studio) si hanno a disposizione in un blocco di giorni, assegnando una quota di ore per giorno. Un metodo *formato* si occupa di restituire le ore col formato a 12 e non 24. Vi sono poi tutti i metodi virtuali, di cui verra' poi fatto l'override nelle sotto-classi, tenendo conto dei campi dati aggiuntivi. Il distruttore della classe e' virtuale per permettere la distruzioni di oggetti in presenza di puntatori polimorfi. La funzione virtuale *reset* si occupa di riportare lo stato dell'oggetto a una situazione di default. Per quanto riguarda le operazioni "aritmetiche" abbiamo la somma e la sottrazione. Ve ne sono di vario genere, dalle piu' semplici per sommare e sottrarre singoli secondi o minuti o ore, alle piu' complesse come *SommaOrario* e *SubOrario.* Quest'ultime rappresentano una ridefinizione degli operatori di somma e sottrazione tra oggetti di tipo Orario. Si e' deciso di marcarle *const* in virtu' della possibilita' di non provocare side-effect, ma di restituire un nuovo oggetto (risultato) da assegnare e/o confrontare. Tutte quest'ultime operazioni sono ovviamente virtuali (e ridefinite nelle sottoclassi) per permettere, tramite late-binding, la giusta scelta dell'operazione per il tipo effettivamente puntato dall'eventuale puntatore polimorfo. Poiche' questa classe e' stata pensate per rappresentare il tempo in un solo giorno, qualsiasi operazione aritmetica che faccia eccedere (per eccesso o per difetto) le 24 ore, produrra' un risultato equivalente al tempo ecceduto (es: 20:00:00 + 7:00:00 = 3:00:00).

- La prima sottoclasse di Orario e' **Data**, i cui oggetti rappresentano un orario digitale ereditato, con l'aggiunta del giorno, del mese, dell'anno (tre numeri interi) e dell'emisfero (Boreale o Australe) in cui ci si trova. Quest'ultimo campo dati viene realizzato tramite un valore booleano (per i due possibili stati). Anche in questa classe vi sono dei costruttori per ogni eventualita' (uno di default e uno con 7 parametri di cui alcuni posti di default) e un costruttore di copia. Il distruttore, essendo stato reso virtuale nella classe base, rispettera' questa proprieta' lungo tutta la gerarchia. Tutte le operazioni di interrogazione di Orario vengono ereditate e ne vengono aggiunte di piu' specifiche (*getGiorno, getMese, getAnno, getEmisfero).* Gli operatori di stampa e di confronto vengono riscritti per tener conto dei nuovi campi dati. Dei metodi virtuali di Orario viene fatto override per tener conto dello sforamento in altri giorni (o mesi o anni) dovuto all'aggiunta o sottrazione di quantita' di tempo. Anche il metodo *reset* si adatta ai nuovi campi dati, per cui abbiamo scelto come data di default, l'1 gennaio 2000. Vi sono altri metodi aggiuntivi che abbiamo dovuto marcare virtuali poiche', provocando side-effect, vi sara' bisogno di gestire eventuali modifiche a cascata nelle sottoclassi. Di questi ultimi fanno parte metodi di semplice modifica (*cambiaGiorno, cambiaMese, cambiaAnno)* come metodi piu' complessi come *avanzaUnGiorno, avanzaTotGiorni, indietroUnGiorno e indietroTotGiorni.* Questi metodi si occupano appunto di operare uno shift temporale in giorni, tenendo sempre conto dell'eventuale sforamento in altri mesi o anni. Oltre a queste operazioni, vengono offerti metodi di interrogazione della data corrente in relazione a vari fattori. Si presenta quindi la possibilita' di conoscere il giorno della settimana alla data attuale (*giornoSettimana)*, in quale settimane del mese (*settimanaMese)* o dell'anno (*settimanaAnno)* si e', quanti giorni sono trascorsi dall'inizio dell'anno (*countGiorni) e* in quale stagione si e' (*stagione).* E' presente un metodo statico *bisestile* che appunto conferma o meno se l'anno passato come

parametro, sia bisestile o no. Un ulteriore metodo *giorniMese* permette di conoscere quanti giorni ci sono nel suddetto mese. Vi e' inoltre un metodo *durata* che restituisce la differenza in giorni tra la data attuale e una data passata come parametro.

- La classe Data ha due sottoclassi distinte. La prima e' **FusoOrario**, i cui oggetti rappresentano tutto cio' che e' una Data, con l'aggiunta della fascia oraria in cui ci si trova. Questo aspetto viene rappresentato da due campi dati: un intero per la zona e una `static std::map` per le fasce. In questa `map` le chiavi sono degli interi che vanno da -12 a +12 (0 compreso), mentre i valori sono dei `char` che rappresentano le varie fasce in codifica militare. Vi sono anche in questo caso, vari costruttori utili ad istanziare l'oggetto con campi dati opportuni. Tutti i metodi e le operazioni, virtual e non, di Data, vengono ereditate e non ridefinite. Le uniche di cui viene fatto effettivamente override sono *SommaOraio* e *SubOrario* per motivi pratici, in quanto l'oggetto interno (che verra' poi restituito) deve essere costruito di copia con il costruttore corrispondente della classe corrente. In questa classe il metodo virtuale *reset* viene corretto per riportare la fascia a 0 (Greenwich). Vengono forniti due metodi di ispezione dell'oggetto (*getFascia* e *getZona)*. Le operazioni aggiuntive in questa classe corrispondo alla possibilita' di conoscere il fuso in una determinata zona del mondo (metodo statico *dimmiFuso),* di conoscere la differenza di fuso tra la zona corrente e un'altra passata per paramento (*fusoDiff) e* di aggiungere (*sommaFuso)* o di sottrarre (*togliFuso)* ore al fuso corrente, provocando il ricalcolo della fascia attuale, come se si stesse viaggiando.

- La seconda sottoclasse di Data e' **Luna**, i cui oggetti rappresentano una data, di cui si conosce anche la fase attuale della luna. La fase corrente viene mantenuta in memoria da una `string` che viene assegnata nel costruttore ed e' ispezionabile dal metodo *getFase*. Tutta la classe si basa su un algoritmo specifico atto a calcolare i giorni trascorsi dall'ultimo novilunio (metodo presente nella classe, chiamato *ultimaNuova)*. Questo algoritmo si basa sul calcolo dell'epatta lunare dell'anno corrente (metodo *epatta)* che rappresenta "l'eta' della luna". Con opportuni aggiustamenti ed altri dati ricavati dalla data corrente, e' possibile stabilire con esattezza in quale fase lunare ci si trovi e quanti giorni sono trascorsi tra le varie fasi. Per questo, la classe offre metodi sia di interrogazione precisa (come *piena, crescente, nuova, calante)* che ritornano un semplice booleano di conferma alla richiesta, sia metodi piu' precisi per conoscere quanti giorni mancano (*prossimaPiena* e *prossimaNuova)* o sono trascorsi (*ultimaNuova* e *ultimaPiena)* da una particolare fase lunare. Di tutti i metodi virtuali presenti in Orario e Data viene fatto override per gestire l'eventuale cambio della data corrente e, di conseguenza, della fase lunare.

- Ogni classe della gerarchia utilizza un'eccezione user-defined di nome **Overflow_error**. Questa eccezione riguarda l'inserimento di valori (di solito interi) non corretti. Essa viene lanciata ogni qual volta, in qualsiasi classe della gerarchia, si voglia istanziare un oggetto utilizzando come parametri valori errati: per Orario, ad esempio, non e' possibile avere un oggetto con le ore maggiori di 24 (o minori di 0) o i minuti e secondi maggiori di 59 (o minori di 0); per Data, invece, l'anno non deve essere inferiore al minimo di 1, i mesi devono essere compresi tra 1 a 12 e i giorni non possono superare gli effettivi giorni del mese corrente (calcolati con apposito metodo) ne essere inferiori a 1. Questa eccezione viene lanciata inoltre da alcune operazioni per motivi logici: nella somma ad esempio non ha senso passare un valore negativo, in virtu' della presenza di un'operazione di sottrazione.

4. Progettazione GUI

Per la GUI e' stato adottato il pattern "Model-View-Controller" con libreria Qt. Questo pattern si basa sull'idea di una View (GUI) e di un Model (dati) reciprocamente indipendenti, dove l'intermediario tra utente e dati e' rappresentato dal Controller. Quest'ultimo, ad una chiamata dalla View, interroga il Model (che a sua volta utilizza la gerarchia di tipi), elaborando la risposta e ritornandola alla View chiamante (e quindi all'utente).

- Il Model include la gerarchia di tipi e utilizza come campo dati un puntatore Orario polimorfo che, durante tutta l'esecuzione del programma, puntera' a un oggetto di tipo orario oppure di uno qualsiasi dei suoi sottotipi. Per questo motivo, in alcuni metodi e' stato necessario utilizzare il dynamic_cast al tipo desiderato per permettere la giusta scelta del metodo in fase di compilazione (per metodi della gerarchia non virtuali). Per tutti gli altri metodi, tramite late-binding (grazie alla virtualita' del metodo nella gerarchia), viene usato semplicemente il puntatore polimorfo.
- Il Controller include il Model e ha proprio come campo dati un puntatore a Model. Ogni metodo del controller si occupa di ricevere dei parametri dalla View, elaborandoli in modo tale da poterli trasmettere al Model, chiamando uno dei suoi metodi. Poiche' vi e' la possibilita' che l'utente inserisca dati non corretti e che questi vengano trasmessi al Model, vengono qui gestite eventuali eccezioni ed elaborate in modo tale da non far propagare le suddette fino alla View o, peggio, al Main. Il tipo di ritorno del metodo in cui si gestisce un'eccezione si occupa di rappresentare la riuscita o meno dell'operazione chiamata dall'utente (ad esempio ritorna False o 0 se non e' andata a buon fine).
- La View rappresenta l'interfaccia grafica offerta all'utente. Include il Controller e ne ha un puntatore come campo dati per potersi collegare ad esso e chiamarlo dopo un input dell'utente. Tutti gli altri campi dati sono puntatori agli eventuali pulsanti, label, box e layouts che vengono poi creati effettivamente nel costruttore della View.

5. Descrizione GUI

All'apertura del programma, l'utente si trova di fronte a 4 pulsanti attivi di istanziazione al tipo desiderato. Da sinistra a destra troviamo Orario, Data, FusoOrario e Luna. Alla pressione di uno dei suddetti tasti, appare un form da compilare in cui e' possibile inserire le specifiche del tipo che si intende creare. Alcuni di questi campi sono obbligatori, altri, se non inseriti, saranno posti a 0. A seconda del pulsante selezionato, si attiveranno le varie operazioni, legate sia al tipo attuale sia a quelli da cui eredita. Per Orario, oltre alla visualizzazione in formato *hh:mm:ss*, vi sono un pulsante per sommare, uno per sottrare (ore, minuti e secondi) e uno ulteriore per cambiare la visualizzazione dell'ora impostata. Se il tipo corrente e' Data, verra' visualizzata in formato *dd:mm:yyyy.* Qui, oltre alle operazioni di Orario, si aggiunge la possibilita' di sommare e sottrare giorni, mesi ed anni. Le operazioni proprie sul tipo Data che si basano sulla data attuale sono la possibilita' di sapere la stagione, il giorno della settimana e in quale settimana del mese o dell'anno ci si trova. Vengono inoltre offerte operazioni come il confronto tra date (una durata espressa in giorni) e il calcolo della quantita' di ore di studio che intercorrono tra la data attuale e una data inserita. Alla pressione di FusoOrario, rimangono attive tutte le operazioni di Orario e Data e vengono attivate nuove funzioni (tra cui anche la visualizzazione della zona e della fascia attuale). Queste comprendono sia la somma di ore al fuso corrente (quindi come se si viaggiasse verso Est), sia la sottrazione (viaggiare verso Ovest), ma anche la possibilita' di conoscere il fuso presente nella zona specificata o la differenza che separa il fuso corrente da uno di destinazione. L'abilitazione di un oggetto Luna, disattiva innanzitutto le operazioni proprie di FusoOrario (mantenendo pero' quelle di Orario e Data), attivando nuove funzionalita' esclusive e visualizzando a schermo un'immagine esplicativa della fase lunare attuale. Le funzionalita' comprendono la possibilita' di calcolare quando sono state o saranno le fasi di novilunio e piena.

6. Suddivisione del lavoro progettuale

L'intero progetto e' stato pensato e svolto in ogni sua fase contemporaneamente da entrambi, sia in fase di progettazione che in fase di codifica. L'unica divisione di compiti e' stata fatta per due classi della gerarchia di tipi dove io, Francesco Sacchetto, mi sono occupato della stesura e realizzazione dei files luna.h e luna.cpp e, Nicola Cisternino, si e' occupato di fusoorario.h e fusoorario.cpp. Per tutto il resto del progetto, abbiamo sempre lavorato in coppia sulle stesse parti di codice, fatta eccezione per lo studio individuale della documentazione da utilizzare.

Ore utilizzate:

- Analisi preliminare: 4 ore
- Progettazione modello e GUI: 15 ore
- Apprendimento libreria Qt: 8 ore
- Codifica modello e GUI: 15 ore
- Debugging: 8 ore
- Testing: 5 ore

- Totale: 55 ore. (Le 5 ore eccedute sono state utilizzate per studiare individualmente i metodi delle API utilizzate nell'interfaccia grafica e le informazioni utili alla logica dei metodi nella gerarchia.)