Relazione progetto programmazione ad oggetti anno 2017/2018
Studente: Bello' Marco
Matricola: 1142874
Sistema operativo di sviluppo: Debian GNU/Linux 9.5 (stretch) x86_64
Versione Qt: 4.8.7 (qt4)
Compilatore: GCC x86 64bit 6.3.0

IL PROGETTO
Lo scopo del progetto è quello di modellare, tramite delle classi opportunamente definite da utente, un computer in tutti i suoi componenti basici e fornire un "giudizio", una score, della macchina in questione basata su una mera valutazione numerica delle caratteristiche dei suoi componenti. La score non è stata limitata ad una scala da zero a cento perchè ciò avrebbe comportato la necessità di porre, a priori, una macchina di "benchmark" a cui rapportare tutte le altre, di fatto rendendo il software obsoleto nel giro di 3/5 anni, oltre che poco estensibile.
Chiaramente il giudizio non è necessariamente rapportabile alla "vita vera" in quanto le caratteristiche tecniche di una macchina non danno mai un quadro completo, tuttavia essendo lo scopo del progetto l'esercizio alla programmazione ad oggetti e non la creazione di una calcolatrice da vendere ad un cliente è stato ritenuto accettabile.

CLASSI

Gerarchia dei componenti

Component:
Classe base astratta, si occupa di fornire un'interfaccia comune a tutte le classi della prima gerarchia del progetto e, in un'ottica di estensibilità , dovrebbe essere la classe base da cui far derivare anche gli eventuali nuovi componenti.
Fornisce un distruttore virtuale, un metodo *getScore( )* e un metodo *getPrint( )*, che verranno implementati dalle successive classi derivate da Component.
getScore( ) ha lo scopo di ritornare un giudizio numerico di ogni componente del computer, non è stato usato il codice polimorfo perchè la gerarchia di componenti ha il solo scopo di essere implementata in una successiva gerarchia di macchine (computer, laptop ecc) come campi dati "singoli", non acceduti tramite puntatore, quindi si è preferito favorire l' efficienza lasciando il metodo non virtuale.
getPrint( ) è un metodo di funzionalità che ritorna come QString delle informazioni riguardanti i singoli componenti, una sorta di metodo *toString( )* di Java. Non è stato posto virtuale per le stesse motivazioni di *getScore( )*.

BaseComponent:
Classe astratta derivata pubblicamente da Component che fa da superclasse per tutti i componenti "base", ovvero sempre presenti in qualsiasi macchina, che siano pc fissi, portatili o telefoni cellulari.
Come campi dati ha clockspeed e memory_dimension, che tecnicamente sono presenti in tutti i componenti delle classi figlie.
Rende disponibili *getClock( )* e *getMem( )* per le classi figlie, ma per mantenere l' information hiding questi metodi sono protected.
Rende inoltre disponibili dei metodi per modificare i campi dati, nello specifico *setMemory(double mem)* e *setClock(int clk)* prendono un valore in input tramite i parametri formali e lo assegnano ai campi dati corrispondenti;
modifyMemory(double mem) e *modifyClock(double clk)* invece sommano i valori di input ai campi dati e ritornano, nel caso di un tentativo di modifica illegale, una stringa di errore, oltre che stamparla su std::cerr.
BaseComponent rende inoltre disponibile un costruttore a zero, uno e due parametri.
Inoltre BaseComponent ha l'operatore di output operator<< ridefinito.

CentralProcessingUnit:
Classe concreta derivata pubblicamente da BaseComponent, eredita i suoi campi dati e ci aggiunge un intero che rappresenta il numero dei cores, un booleano che rappresenta la presenza o meno dell'hyperthreading e un campo dati statico che aumenta di dieci punti percentuali la score della CPU se essa ha hyperthreading.
Implementa *getPrint( )* e *getScore( )* e fornisce dei metodi di utilità per modificare i propri campi dati: come in BaseComponet *setCores(int cores)* assegna il valore di input al campo dati, mentre *modifyCores(cores)* lo somma e ritorna, nel caso di un tentativo di modifica illegale, una stringa di errore, oltre che stamparla su std::cerr.
SetHyperT(bool hy) invece prende un booleano in input e lo assegna direttamente al campo dati hyperthreading.
Rende disponibile il costruttore a zero, uno, due, tre o quattro parametri e ha l'operatore di output operator<< ridefinito.

GraphicsProcessingUnit:
Classe concreta derivata pubblicamente da BaseComponent, eredita i suoi campi dati e ci aggiunge un intero per il numero di cores (come CPU), un intero per la bandwidth e un booleano che rappresenta la presenza o meno di una GPU dedicata.

Implementa i metodi *getPrint( )* e *getScore( )*, rende disponibili *setCores(int cores)* e *setBand(int bandwidth)* che assegnano i valori di input ai campi dati, mentre *modifyCores(int)* e *modifyBand(int)* li sommano e ritornano, nel caso di un tentativo di modifica illegale, una stringa di errore, oltre che stamparla su std::cerr.
Il metodo *presenceGPU(bool)* prende un booleano in input e lo assegna direttamente al campo dati gpuPresence, mentre il metodo *thereIsGPU( )* ritorna true se la gpu è presente, false altrimenti.
Rende disponibile il costruttore a zero, uno, due, tre, quattro o cinque parametri e ha l'operatore di output operator<< ridefinito.

RandomAcessMemory:
Classe concreta derivata pubblicamente da BaseComponent, eredita i suoi campi dati.
Implementa i metodi *getPrint( )* e *getScore( )*.
Rende disponibile il costruttore a zero, uno o due parametri e ha l'operatore di output operator<< ridefinito.

SecondaryMemory:
Classe concreta derivata pubblicamente da BaseComponent, eredita i suoi campi dati e ci aggiunge due interi che rappresentano le velocità di lettura e scrittura.
Implementa i metodi *getPrint( )* e *getScore( )*.
setWrite(int) e *setRead(int)* assegnano direttamente i valori in input ai campi dati, mentre *modifyWrite(int)* e *modifyRead(int)* li sommano e ritornano, nel caso di un tentativo di modifica illegale, una stringa di errore, oltre che stamparla su std::cerr.
Rende disponibile il costruttore a zero, uno, due, tre o quattro parametri e ha l'operatore di output operator<< ridefinito.

Screen:
Classe concreta derivata pubblicamente da Component, si differenzia dalle classi precedenti in quanto non è un "componente base" (non ha bisogno di clockspeed e memory_dimension) ed è per questo che non deriva da BaseComponet.
Siccome è uno schermo i suoi campi dati rappresentano la risoluzione orizzontale e verticale (int pixels) e la dimensione diagonale (in pollici).
Rende disponibili i metodi *setRes(int, int)* e *setDim(double)* che si occupano rispettivamente di assegnare i campi dati per la risoluzione e per la dimensione, inoltre rende disponibili i metodi *getResW( ), getResH( )* e *getDim( )* che vanno a inficiare sull' information hiding ma servono ad una delle classi della seconda gerarchia.

Gerarchia delle macchine

Computer:
Classe base astratta della seconda gerarchia, ovvero della gerarchia delle macchine (la precedente era la gerarchia dei componenti).
I suoi campi dati sono CPU, GPU, RAM e Memoria Secondaria (HDD o SSD) e il prezzo.
Rende disponibile diversi metodi di *set* e *modify*, corrispondenti a tutti i *set* e *modify* dei campi dati ovvero delle classi della gerarchia dei componenti che, rispettivamente, assegnano i campi dati oppure li sommano e ritornano, nel caso di un tentativo di modifica illegale, una stringa di errore, oltre che stamparla su std::cerr.
Sono inoltre presenti *setPrice(double)* e *modPrice(double)* che si comportano esattamente come i *set* e *modify* di cui sopra.
Computer rende inoltre disponibili due diversi metodi virtuali di Print, ovvero *Print( )* e *getPrint( )*: il primo stampa in formato "leggibile" da utente (leggibile inteso come fosse il flag --human-readable di alcuni comandi di Linux) le caratteristiche della macchina su std::cout, mentre il secondo ritorna una stringa di testo che rappresenta la macchina (di fatto ritorna la stessa stringa che *Print( )* stamperebbe sullo standard output).
Questi metodi sono virtuali perchè le macchine vengono gestite tramite una lista che punta ad oggetti di tipo Classe (Tower, Laptop e Smartphone) sulla heap, e quindi è possibile utilizzare lo stesso metodo Print con chiamata polimorfa per ottenere la Print corretta rispetto al tipo di macchina che si sta correntemente utilizzando. Nello specifico la chiamata risulterebbe così: *(*it)->Print( );* oppure *QString s=(*it)->getPrint( );*
Computer rende inoltre disponibile il metodo virtuale ***Score( )*** che si occupa di ritornare, con chiamata polimorfa analoga a quella per le Print, il punteggio della macchina corrente. In pratica avendo un iteratore che punta ad una lista con oggetti dei tre diversi tipi di macchine, per ottenere la Score corretta sarà sufficiente fare *(*it)->Score( );* e verrà scelto il metodo corretto corrispondente al tipo puntato da *it.
Computer ha il distruttore virtuale e il costruttore a zero, uno, due, tre, quattro e cinque parametri.

Tower:
Classe concreta derivata pubblicamente da Computer, non aggiunge alcun campo dati e si limita ad implementare *Score( )*.
Rende disponibile un costruttore a zero, uno, due, tre, quattro e cinque parametri.

Laptop:
Classe concreta derivata pubblicamente da Tower, aggiunge ai propri campi dati uno schermo di tipo classe Screen, una batteria e il peso (quest' ultimi rappresentati come interi). La batteria è in milli Ampere ora mentre il peso è in grammi. Rende disponibili i metodi di set e modify con il solito comportamento (assegnazione per i primi, somma con eventuale stringa e stampa su standard error dell'errore in caso di valori illegali) per il settaggio e modifica di batteria e peso, mentre rende disponibile solo il metodo di set *setResolution(int,int)* per la risoluzione dello schermo (sembrava infatti un po' inutile l' idea di "sommare" pixel per la risoluzione).
Tower inoltre implementa i metodi virtuali *Print( ), getPrint( )* e *Score( )* e rende disponibile un costruttore da zero a otto parametri.

Smarphone:
Classe concreta derivata pubblicamente da Laptop, aggiunge solo un campo dati ovvero la generazione del telefono (1G, 2G, 3G, 4G...).
Implementa i metodi virtuali *Print( ), getPrint( )* e *Score( )* e mette a disposizione il metodo *setGen(int)* che assegna al campo dati corrispondente la generazione presa da input tramite parametro formale.
Smartphone rende inoltre disponibile il costruttore da 0 a 9 parametri.

GUI

KalkWidget.cpp:
E' la classe che si occupa, tramite i suoi slot, di far funzionare la gui. Utilizza una ***std::list<Computer*>*** per gestire run time tutte le varie macchine che l' utente potrebbe voler utilizzare e in ogni momento, nella mainWindow dell' applicazione, sono visualizzate tutte le informazioni riguardanti la macchina corrente. Appena al di sopra si trova una finestra per la visualizzazione degli errori run time.
KalkWidget si avvale inoltre di tre metodi definiti da utente: *at( )* che ritorna la posizione della macchina corrente, *laptopVisibility(bool)* e *smartphoneVisibility(bool)* che si occupano invece di rendere visibili o non visibili quelle interfacce della gui dedicate solo ed esclusivamente all' utilizzo con una macchina di tipo Laptop o Smartphone.

Come usare la gui:
I bottoni Previous e Next in alto permettono di spostarsi tra le macchine attualmente in uso, mentre i bottoni New Tower, New Laptop e New Smartphone creano una nuova macchine rispettivamente di tipo Tower, Laptop o Smartphone.
Il bottone Remove Machine elimina la macchina corrente, mentre il bottone Get Score stampa la score della macchina corrente.
Al di sotto troviamo tutte le interfacce per modificare e settare le varie caratteristiche tecniche delle macchine in uso, prima fra tutte il prezzo e poi sotto a seguire per CPU, GPU, RAM, MEM, Schermo, Peso, Batteria e Generazione telefonica. I bottoni set assegnano il valore scritto nella lineEdit alla loro sinistra al campo dati corrispondente, mentre i bottoni add la sommano.
La risoluzione va settata scrivendo nella prima lineEdit la risoluzione orizzontale, nella seconda lineEdit (dopo la "X") la risoluzione verticale.
I bottoni per hyperthreading cpu e presenza gpu sono caratterizzati da non avere una lineEdit, ma semplicemente premere, ad esempio, il bottone YES di fianco al label Presence nella sezione GPU sta a significare che la GPU è presente.
Analogamente premere il bottone NO di fianco al label Hyperthreading nella sezione CPU comporterà avere una CPU senza hyperthreading.

EXTRA

Cos' è inputterminal.cpp:
Si tratta di un'interfaccia alternativa alla gui per testare il programma, utilizzata in fase "beta" prima ancora di iniziare a pensare alla realizzazione di KalkWidget. I comandi necessari per il suo utilizzo si trovano alla fine del file "inputterminal.cpp" sotto forma di commento.

Gestione degli errori:
La gestione degli errori segue due strade differenti: per quanto riguarda i *set* i controlli vengono fatti direttamente da KalkWidget in quanto si tratta solo di controllare che i valori inseriti da input non siano <=0. Per quanto riguarda i *modify* o comunque gli *add* il controllo deve avvenire "back end" ovvero nella gerarchia delle classi Componente per evitare di dover venire meno all' information hiding "passando" a KalkWidget i valori dei campi dati per eseguire i controlli. Le classi Componente si occupano poi di stampare su std::cerr un messaggio di errore e di ritornarlo sotto forma di QString.
Esempio: se ho una CPU con 2 cores e voglio sottrarne 3 (quindi sommare -3) l'unico modo che ho per sapere che otterrò un numero di cores <=0 e' conoscendo il valore del campo dati num_physical_cores.

Perchè set e modify non sono virtuali:
La motivazione è molto semplice: si tratta di metodi che vengono sempre chiamati direttamente su un oggetto senza l'ausilio di un puntatore, infatti le classi della gerarchia Component "vivono" solo come campi dati degli oggetti della gerarchia Computer (che invece sfrutta il codice polimorfo). Ancora più nello specifico: essi come campi dati non sono pensati come una lista di componenti perchè, tranne in rari e costosi casi, un computer avrà uno e un solo processore, una e una sola scheda video, un banco di ram, una determinata quantità di memoria secondaria totale.
Si sarebbe potuto fare un sistema a lista, ma sembrava più artificioso che sensato, e si è preferito mantenere il codice efficiente e non inutilmente complesso.

Come Compilare:
Entrare nel folder Progetto_gui ed utilizzare il file .pro Progetto_gui.pro

cd Progetto_pao1718/Progetto_gui
qmake Progetto_gui.pro
make

Come usare il progetto in Java:
cd Progetto_pao1718/java/esecuzione
*javac *.java*
java Use

Monte Ore:
Progettazione e scrittura delle gerarchie Component e Computer: 30 ore
Traduzione in java: 10 ore
Studio di Qt, progettazione ed esecuzione della gui: 10 ore
Relazione: 3 ore
Debug: 7 ore

Il monte ore non è stato rispettato perchè c' è stato, all'inizio, un errore di valutazione riguardo il tempo che il progetto avrebbe potuto richidere.