Vittorio Santagiuliana 1170542

Relazione progetto di Programmazione ad oggetti

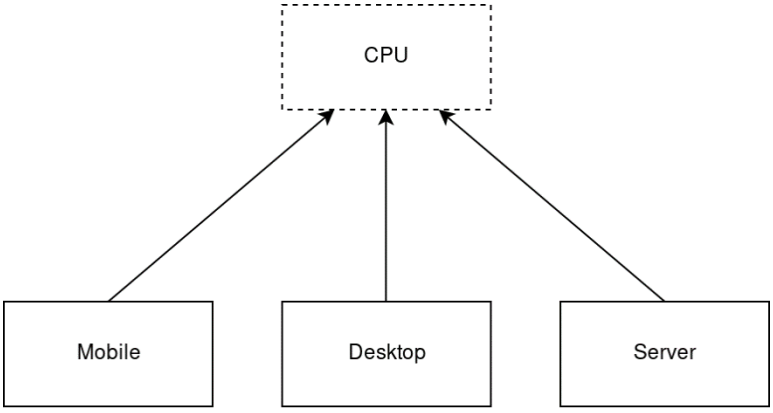
Note di compilazione

L'ambiente di sviluppo utilizzato per questo progetto è Qt Creator, su Linux Manjaro 18.0.4 con framerwork Qt 5.13.0 e compilatore gcc 9.1.0. Viene incluso il file `.pro` necessario per la compilazione tramite I comandi `qmake` e `make`.

Introduzione

L'applicativo CPU-Qatalog si pone l'obiettivo di gestire un catalogo di processori basati sul set di istruzioni x86, suddividendoli nelle sottocategorie *desktop*, *mobile* e *server*, e consente l'aggiunta, la rimozione e la modifica dei processori in lista.

Gerarchia



Classe base astratta: CPU

Rappresenta un generico processore, con caratteristiche quali il produttore, il nome del modello, il numero di core fisici e l'architettura.
La classe è dotata di una mappa statica dei produttori, così da poter avere un controllo sui possibili valori che il campo produttore possa assumere.

Classe Mobile

Derivata da CPU, rappresenta i processori per dispositivi mobili, dotati di meno core fisici e con un consumo limitato.
La classe contiene, come tutte le altre classi concrete nella gerarchia, una mappa statica dei socket di questa tipologia di processori.

Classe Desktop

La classe desktop rappresenta i processori per pc fissi; rispetto a processori per dispositivi mobili hanno un TDP (Thermal Design Power) più alto ed eventuale supporto a memorie ECC (Error-Correcting Code)

Classe Server

Infine i processori della classe server sono quelli dotati del maggior numero di cores e con supporto a memorie ECC.

Polimorfismo

La classe base astratta fornisce i seguenti metodi virtuali puri:

- `virtual const std::string & getSocket() const = 0` : è il getter del campo socket, restituisce la stringa associata al valore del campo socket. Poiché ognuna delle classi contiene una mappa statica dei socket, la funzione viene reimplementata nelle classi derivate così da poter restituire il nome del socket, presente nella mappa.
- `virtual void setSocket(const int) = 0` : setter del socket, verifica la sua presenza nella mappa prima dell'assegnazione, in caso negativo solleva un'eccezione.
- `virtual void setCoreCount(const int) = 0` : setter del numero di cores, controlla che il valore sia minore od uguale ad un parametron statico nella classe, altrimenti solleva un'eccezione.
- `virtual void setTdpRating(const int) = 0` : come il setter dei cores, effettua un controllo prima dell'assegnazione.
- `virtual void setEccMemorySupport(const bool) = 0` : setter per il supporto di memorie ECC, non disponibile nelle classi mobile e server.
- `virtual CPU * clone() const = 0` : metodo per la clonazione polimorfa, effettua la copia profonda dell'oggetto di invocazione e ne ritorna un puntatore.

Qontainer

La gestione dei dati viene affidata alla classe template Qontainer<T>, che funge da contenitore di oggetti di tipo T e consente di effettuare operazioni di inserimento, rimozione sia in coda che in una posizione specificata tramite un iteratore.
Per quanto riguarda l'implementazione, il contenitore funge da array a dimensione variabile utilizzando l'allocazione dinamica di array di oggetti T.
I pregi di questa implementazione sono un tempo costante per l'accesso, e un tempo costante ammortizzato per aggiunta e rimozione in coda, che sono operazioni effettuate frequentemente dalla vista.
Le operazioni più dispendiose sono l'inserimento e la rimozione in posizioni arbitrarie, che richiedono lo spostamento di tutti gli elementi successivi a quello interessato.

Modello

Il modello dati si appoggia su un Qontainer di deepPtr, classe di supporto che funge da puntatore, offrendo le funzionalità di copia, assegnazione e distruzione profonda.

Il modello si occupa anche della serializzazione e deserializzazione degli oggetti nel Qontainer in formato XML appoggiandosi alle librerie Qt QXmlStreamReader e QXmlStreamWriter.

Framework Qt

Il progetto è stato sviluppato seguendo il design pattern model-view ed è stata utilizzata una QTableView che attraverso la classe TableModel, ereditata da QAbstractTableModel, ottiene i dati dal modello.

L'interfaccia comprende anche le finestre addDialog ed editDialog, rispettivamente per l'inserimento e modifica dei dati.

Estensibilità e manutenibilità

Parti della vista (ad esempio la finestra addDialog) e del modello (metodi di serializzazione) hanno un'implementazione dipendente dalle classi presenti nella gerarchia, questo causa un peggioramento per la manutenibilità dell'applicazione nel caso di modifica o aggiunta di una classe.
L'utilizzo di mappe nelle classi della gerarchia garantisce per lo meno un livello di separazione tra la vista e i valori dei singoli campi, sui quali vengono applicate limitazioni ottenute direttamente dal modello. Così facendo, l'aggiunta di un nuovo produttore o di un nuovo socket non richiede nessuna modifica oltre all'inserimento nel modello.
Allo stesso tempo la modularità e separazione dei componenti utilizzati facilitano la sostituzione o modifica di componenti nell'applicazione, ad esempio con modifiche minime al codice è possibile sostituire il Qontainer con un qualsiasi altro contenitore.

Tempo impiegato

- Analisi realtà da modellare: 2h
- Implementazione modello: 8h
- Implementazione Qontainer e deepPtr: 8h
- Apprendimento libreria Qt: 10h
- Implementazione GUI e connessione vista-modello: 16h
- Debugging: 10h.