Universita' degli Studi di Padova

Corso di Programmazione ad oggetti

GenetiKalk

Relazione progetto Kalk
A.A. 2017/2018

Autori	*Matricola*
A cura di: Trevisin Andrea	1144684
Partner: Segna Carlotta	1123208

8 luglio 2018

Indice

1 Abstract

Il progetto si propone di sviluppare un calcolatore in ambito biologico, in grado effettuare operazioni su biomolecole (in particolare per quanto riguarda DNA, RNA, amminoacidi e proteine).
Durante la progettazione è stata fatta la scelta di adottare il design pattern MVC: l'applicativo, contenente un modello logico che descrive la realtà di interesse, mette a disposizione dell'utente un interfaccia semplice ed intuitiva (in stile calcolatrice) per visualizzare tale modello ed interagirvi, i cui input sono gestiti internamente da un controller separato.
Nella realizzazione si è scelto di procedere in modo tale che l'applicativo possa essere potenzialmente esteso con ulteriori tipi di calcolo: è possibile, volendo, definire facilmente ulteriori rappresentazioni di biomolecole o raffinare quelle già presenti tramite le gerarchie implementate.

2 Compilazione

Con il progetto viene fornito il file `GenetiKALK.pro`; è sufficiente eseguire il comando `qmake` nella stessa cartella (directory principale del progetto).

3 Suddivisione del lavoro progettuale

Io mi sono occupato della realizzazione di:

- File C++ (header & source): Monomero, Nucleotide, sequenzaDNA, sequenzaRNA, intera directory controller, intera directory view (meno ProWidget), exceptionhandler
- File JAVA: classi DNA e RNA
- Progettazione, debugging, testing: definizione generale in parti uguali, poi le responsabilità sono state suddivise in accordo alla realizzazione

La mia partner di progetto si é occupata dei file non menzionati.

4 Introduzione teorica

Trattandosi di un progetto legato specificatamente alla biologia, ed in particolare alla biochimica, di seguito verrà fornito un breve glossario per aiutare la comprensione della realtà di interesse:

- **Monomero**: semplice molecola, o famiglia di molecole, in grado di formare legami con altre molecole identiche (o appartenenti alla stessa famiglia) dando luogo a "catene" di molecole simili (un po' come dei mattoncini Lego).
- **Polimero**: molecola complessa risultante dall'unione di due o più monomeri.
- **Nucleotidi**: particolare famiglia di monomeri, compongono DNA e RNA. Esistono cinque nucleotidi: adenosina, timidina (esclusiva al DNA), citidina, guanosina e uridina (esclusiva al RNA). Ogni nucleotide può legare "lateralmente" altri nucleotidi, formando catene, e poi "frontalmente" il/i proprio/i nucleotide/i complementare/i, formando catene doppie.
- **DNA**: acronimo di acido desossiribonucleico, polimero situato nel nucleo delle cellule. Si trova sotto forma di filamento singolo o di doppio filamento avvitato ad elica; nel doppio filamento, ogni nucleotide è appaiato al suo nucleotide "complementare".
- **RNA**: acronimo di acido ribonucleico, polimero che si occupa di trascrivere il contenuto del DNA (sfruttando il meccanismo dei nucleotidi complementari) permettendo la sua "lettura" nella sintesi di proteine.
- **Amminoacidi**: particolare famiglia di monomeri, costituenti delle proteine. Esistono 20 amminoacidi; nella sintesi di una proteina, l'RNA viene letto a gruppi di 3 nucleotidi ("codoni") associando ogni gruppo ad un particolare amminoacido.
- **Proteine**: famiglia di polimeri complessi costituiti da amminoacidi e presenti in ogni parte di un essere vivente, vengono sintetizzate traducendo l'RNA (che a sua volta trascrive il contenuto del DNA) ed assemblando gli amminoacidi prodotti.

5 Descrizione delle gerarchie

5.1 Gerarchia del modello

Posto che la realtà descritta prevede molecole realmente esistenti, nucleotidi e amminoacidi sono istanziati una sola volta in un file di costanti mediante (`gen_costants.h -> gen_costants.cpp`).

5.1.1 class IPolimero

IPolimero è una classe base astratta con funzione di interfaccia, atta a permettere e facilitare la definizione di puntatori polimorfi ad oggetti derivati dalla classe templatizzata ***Polimero<T>***. La classe dichiara i seguenti metodi virtuali puri: **updateInfo()**, **toString()** e **concat(IPolimero*)**. L'intenzione è quella di limitare le chiamate polimorfe attraverso un puntatore di tipo IPolimero a funzioni non dipendenti dal tipo di istanziazione dell'oggetto puntato e che non modificano l'array di elementi interno.

5.1.2 class Polimero<T> : public IPolimero

Classe templatizzata che implementa la classe interfaccia, rappresenta un generico polimero.
Nel costruttore un assert forza l'istanziazione con T in subtyping a ***Monomero***.
Campi dati:

- `QVector<T*> _elems`: vettore di puntatori ai monomeri che compongono il polimero.
- `int _length`: lunghezza del polimero.
- `double _mass`: massa del polimero.

Metodi:

- **Metodi di get e set**, **costruttore di copia profonda**, **distruttore profondo**.
- `virtual void addElem(const T&)`: aggiunta di un monomero in coda a `_elem`.
- `virtual void delElem()`: eliminazione di un monomero in coda a `_elem`.
- `virtual Polimero<T>* concat(IPolimero* const) const`: concatenazione di due polimeri. Il controllo sul tipo di istanziazione viene effettuato tramite `dynamic_cast` e gestito da un'eccezione.
- `virtual double calcMass()`: calcolo della massa del polimero.
- `virtual QString toString() const`: traduzione della catena di monomeri in un valore `QString`.

5.1.3 class seqDNA: public Polimero<Nucleotide>

Classe derivata dall'istanza di template ***Polimero<Nucleotide>***, descrive una sequenza di DNA.
Campi dati:

- `bool _isSense`: booleano che descrive la direzione della catena di DNA.
- `seqDNA* _seqComp` : puntatore ad un oggetto ***seqDNA*** rappresentante l'eventuale filamento complementare.

Metodi:

- **Metodi di get e set**, **costruttore di copia profonda**, **distruttore profondo**.
- ridefinizioni di `addElem(...)`, `delElem()`, `concat(...)`, `calcMass()`.
- `bool validate(const Nucleotide&) const`: valida l'inserimento di un nuovo elemento.
- `seqDNA* calcSeqComp()`: calcolo del filamento complementare.
- `seqDNA* toSeqComp()`: scambio con il filamento complementare.
- `void delSeqComp()`: eliminazione del filamento complementare.
- `seqRNA* toRNA() const`: calcolo della corrispondente sequenza di RNA, ritorna un puntatore ad un nuovo oggetto `seqRNA`.

5.1.4 class seqRNA : public Polimero<Nucleotide>

Classe derivata dall'istanza di template ***Polimero<Nucleotide>***, descrive una sequenza di RNA. Campi dati:

- `bool _isSense`: booleano che descrive la direzione della catena di RNA.

Metodi:

- **Metodi di get e set**, **costruttore di copia profonda**, **distruttore profondo** (ereditato da Polimero).
- ridefinizioni di `addElem(...)`, `delElem()`, `concat(...)`, `calcMass()`.
- `bool validate(const Nucleotide &) const`: valida l'inserimento di un nuovo elemento.
- `void splicing(unsigned int, unsigned int)`: effettua un taglio nella sequenza tra gli indici indicati.
- `seqDNA* toDNA() const`: calcolo della rispettiva sequenza di DNA.
- `Proteina* toProtein() const`: calcolo della rispettiva proteina.

5.1.5 class Proteina : public Polimero<Amminoacido>

Classe derivata dall'istanza di template ***Polimero<Amminoacido>***, descrive una proteina.
Campi dati:

- `double _volume`: volume della proteina.

Metodi:

- **Metodi di get e set**, **costruttore di copia profonda**, **distruttore profondo** (ereditato da Polimero).
- ridefinizioni di `addElem(...)`, `delElem()`, `concat(...)`, `calcMass()`.
- `bool validate(const Amminoacido &) const`: valida l'inserimento di un nuovo elemento.
- `double calcVolume()`: calcolo del volume della proteina.
- `bool isSimilar(const Proteina&) const`: determina se la proteina passata per argomento appartiene alla stessa famiglia della proteina su cui è invocato il metodo.

5.1.6 class Monomero

Classe base concreta, descrive un monomero generico. Campi dati:

- `QString _name`: nome del monomero.
- `QString _acr`: acronimo del monomero.
- `double _mass`: massa del monomero.

Metodi:

- **Metodi di get e set**.
- `Monomero* clone() const`: metodo standard di clonazione.
- `bool operator==(const Monomero ) const`: ridefinizione dell'operatore di uguaglianza.

5.1.7 class Nucleotide

Classe derivata, descrive un nucleotide generico.
Campi dati:

- `bool _hasOxy`: tipo di zucchero (true = ribosio, false = desossiribosio).
- `QString _comp`: nucleotide complementare.

Metodi:

- **Metodi di get e set**.
- ridefinizioni di `clone(), operator==`.

5.1.8 class Amminoacido

Classe derivata, descrive un amminoacido generico.
Campi dati:

- `QVector<QString> _codon`: vettore dei codoni associati.

Metodi:

- **Metodi di get e set**.
- ridefinizioni di `clone(), operator==`.
- `double calcPercentage(const QVector<Amminoacido*> &) const`: calcola la percentuale in cui l'amminoacido compone la catena di amminoacidi passata per argomento.
- `int findInSequence(const QVector<Nucleotide*>&) const`: restituisce l'indice in una sequenza dell'inizio del primo codone codificante per l'amminoacido di invocazione.

5.2 Gerarchia della vista

La vista è stata progettata anch'essa con ereditarietà e codice polimorfo; la gerarchia rispecchia quella stabilita nel modello. In particolare sono stati realizzati:

- `class PoliQTabWidget : public QTabWidget`
  classe derivata da QTabWidget, le viene assegnato un `dispatchController`.
- `class PoliQWidget : public QWidget`:
  classe derivata da QWidget, le viene assegnato un `poliController`; contiene ed inizializza gli elementi comuni di interfaccia (display, console log, groupbox e pulsanti per operazioni sulla memoria); implementa i metodi `refreshDisplay()` (aggiorna il display), `operationLog()` (scrive nella console) e `confirmOp(...)` (richiede conferma per un'operazione).
- `class DNA/RNA/ProQWidget : public PoliQWidget`
  classi derivate da PoliQWidget, implementano gli elementi specifici di interfaccia (pulsanti per operazioni, pulsanti di input, layout dei pulsanti).

Volendo implementare un nuovo tipo di polimero, è sufficiente creare una classe derivata di `PoliQWidget` per ottenere già uno schema di interfaccia facilmente personalizzabile.

5.3 Gerarchia del controller

Anche la gerarchia del controller ricalca quella definita nel modello, rendendo disponibili:

- `class dispatchoOntroller : public QObject`
  semplice controller assegnato ad un `PoliTabQWidget` che si occupa del trasferimento di oggetti `IPolimero*` tra `poliController`.
- `class poliController : public QObject`:
  semplice classe astratta, con funzionalità di interfaccia per controller specifici che lo implementino (è stato scelto di non fornire un controller generico). In particolare permette la ricezione di un oggetto trasmesso da un `dispatchController` e il suo inserimento come `_activeObj` in un controller specifico.
- `class DNA/RNA/ProController : public poliController`
  controller specifici per il tipo di widget e parte del modello che gestiscono, costituiscono la parte "meno flessibile" della gerarchia, in cambio di maggior robustezza e meno operazioni di cast. In generale implementano un `_activeObj` come oggetto di operazione attivo, un `_storedObj` come oggetto salvato in memoria e le varie funzioni connesse ai bottoni (i `connect` vengono effettuati alla costruzione del controller), in accordo alle parti di modello e di interfaccia gestite dal controller.

6 Eccezioni

La gestione delle eccezioni è stata realizzata in modo semplice e completo creando una gerarchia di eccezioni (per comodità dichiarata e definita all'interno dello stesso file header) che fanno da wrapper ad un messaggio testuale (un valore `QString`); sotto questo punto di vista la gerarchia ha più un valore descrittivo (leggibilità del codice) che effettivo. Tutte le eccezioni lanciate dal modello sono catturate e gestite dal controller, che impedisce operazioni compromettenti e comunica all'interfaccia eventuali messaggi di errore. Sono state definite ulteriori eccezioni che, in caso di implementazione erronea dell'interfaccia in fase di estensione del progetto, nascondono la MainWindow e producono un messaggio di errore in cout.

7 Polimorfismo

I casi in cui è stato scelto di sfruttare il polimorfismo sono i seguenti:

- `IPolimero*`: la funzione `IPolimero::concat(IPolimero*)`, che ha un parametro polimorfo e viene ridefinita sfruttando la covarianza dei tipi.

- `poliController`, che permette l'assegnazione di un controller specifico ad un widget tramite un puntatore polimorfo in `PoliQWidget` nonché la chiamata polimorfa di `receiveObj(...)` nella trasmissione di un oggetto da parte di un `dispatchCOntroller` senza sapere il tipo del controller di destinazione. In dettaglio, la variabile `receiver`, di tipo statico PoliQWidget e di tipo dinamico risultato della chiamata a `findChild`, chiama la funzione `getController` sul cui risultato (un controller di tipo statico `poliController` e di tipo dinamico dipendente dal tipo dinamico del widget restituito da `findChild`) viene chiamata la funzione polimorfa `receiveObj`(...), nelle varie classi di controller derivate per accettare puntatori polimorfi di tipo statico `IPolimero` e fare un safe cast al tipo dinamico.
- Il metodo `activeObj`() nello stesso controller che permette di implementare il refresh del display in `PoliQWidget` senza sapere il tipo dinamico dell'oggetto rappresentato (dichiarato virtuale in con tipo `IPolimero` in `poliController`, e poi ridefinito nelle classi derivate: la funzione di refresh chiama `activeObj`() su un oggetto di tipo statico `poliController` e tipo dinamico diverso, risultando quindi in una chiamata al metodo ridefinito).
- `PoliQWidget*` nella dichiarazione dei widget da aggiungere a `PoliQTabWidget`, che permette di assegnare in modo polimorfo widget di classe derivata da `PoliQWidget` mantenendo la possibilità di cercarli con `findChild<poliQWidget>` (ulteriori dettagli sono disponibili come commenti al codice).

8 Considerazioni e spunti di miglioramento

Con più ore a disposizione, sarebbe stato possibile ampliare il programma con un file parser per definire le costanti in un file .cfg, una schermata di modifica della precisione della misura di massa, una schermata di settings, styiling tramite QML ed eventualmente lal realizzazione di file di traduzione, implementabili con `tr`.

9 Tempo impiegato

- Analisi preliminare: 3h
- Progettazione modello, GUI, controller: 15h
- Apprendimento libreria Qt: 5h
- Codifica modello, GUI, controller: 28h
- Debugginge e testing: 6h
- **TOTALE**: 57h

Le ore in eccesso sono state causate principalmente dal rifacimento di alcune parti del progetto in corso d'opera.