Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione

Corso di laurea triennale in ingegneria informatica

Progettazione e implementazione di un database per l'integrazione e la formalizzazione di dati relativi alla fosforilazione di proteine.

Relatore: ***Prof.ssa Concettina Guerra***

Correlatore: ***Marco Mina***

Laureando: ***Fabio Marangon***

Anno Accademico: 2010 - 2011

Indice

Indice **2**

1 Introduzione **5**
1.1 Il problema . . . 5
1.1.1 Proteine . . . 5
1.1.2 Fosforilazione . . . 6
1.1.3 Situazione attuale . . . 6
1.1.4 Obiettivi . . . 7
1.2 Lavoro svolto e risultati ottenuti . . . 7
1.2.1 Parsing . . . 7
1.2.2 Mapping . . . 8
1.2.3 Creazione DB . . . 8
1.2.4 Altri risultati . . . 8
1.3 La tesi . . . 9

2 I database di fosforilazione **11**
2.1 Caratteristiche database sorgenti . . . 11
2.1.1 UniProt . . . 12
2.1.2 PTM . . . 15
2.1.3 ELM . . . 17
2.1.4 HPRD . . . 19

3 Il nuovo database *PhosphoDB* **21**
3.1 Caratteristiche nuovo database . . . 21
3.1.1 Schema ER . . . 22
3.1.2 Schema relazionale . . . 28
3.1.3 Normalizzazione . . . 30
3.1.4 Simmetria ed estendibilità . . . 33
3.2 Query di esempio . . . 34
3.2.1 Query #1 . . . 34
3.2.2 Query #2 . . . 35

3.2.3 Query #3 . . . 35
3.2.4 Query #4 . . . 36
3.2.5 Query #5 . . . 37
3.2.6 Query #6 . . . 38

4 Struttura del programma **39**
4.1 Modularità . . . 39
4.1.1 Classi e interfacce . . . 39
4.2 Bottom-Up vs Top-Down . . . 42
4.3 Modulo Parser . . . 43
4.3.1 Species . . . 43
4.3.2 SpeciesHandler . . . 43
4.3.3 PDB . . . 44
4.3.4 PDBHandler . . . 45
4.3.5 UniProtRecord . . . 45
4.3.6 Parser . . . 47
4.4 Modulo Mapper . . . 50
4.4.1 DBLink . . . 50
4.4.2 IDMapper . . . 51
4.5 Modulo Populator . . . 54
4.5.1 DBParameters . . . 54
4.5.2 DBConnector . . . 54
4.5.3 DBHandler . . . 55
4.5.4 DBPopulator . . . 56
4.6 Crezione database . . . 60
4.7 Prestazioni e problemi riscontrati . . . 61

5 Documentazione **63**
5.1 Javadoc . . . 63
5.2 Riutilizzabilità ed estensibilità . . . 67
5.2.1 Esecuzione periodica . . . 67
5.3 Configurazione . . . 68
5.3.1 config.cfg . . . 68
5.3.2 tables.cfg . . . 69

6 Sviluppi futuri **70**
6.1 Front end . . . 70
6.2 Problemi chinasi . . . 71
6.2.1 Collegamenti . . . 71
6.2.2 Parsing approfondito . . . 71
6.3 Problema shift . . . 72

6.4 Problemi minori 73
6.4.1 Fake ID 73
6.4.2 Nomi tabelle 73

Capitolo 1

Introduzione

1.1 Presentazione del problema

1.1.1 Le proteine

Le proteine, sono tra i composti organici più complessi e sono i costituenti fondamentali di tutte le cellule animali e vegetali. Le proteine hanno una organizzazione tridimensionale (struttura) molto complessa a cui è associata sempre una funzione biologica. Lo scheletro delle proteine è costituito da una sequenza di 20 tipi di amminoacidi diversi, cui si aggiungono alcune tipologie speciali di amminoacidi modificati. In una singola proteina non necessariamente sono presenti tutte le tipologie di amminoacidi che invece si trovano in quantità differenti.

Una proteina nel suo complesso è una molecola in cui vengono convenzionalmente distinti vari livelli di organizzazione:

- la *struttura primaria* è formata dalla sequenza specifica degli amminoacidi, dalla catena peptidica e dal numero stesso delle catene e determina da sola il ripiegamento della proteina;

- la *struttura secondaria* consiste nella conformazione spaziale delle catene, ad esempio la conformazione a spirale, o α-elica, o quella planare, o β-strand (foglietto);

- la *struttura terziaria* è rappresentata dalla configurazione tridimensionale completa che la catena polipeptidica assume nell'ambiente in cui si trova.

1.1.2 La fosforilazione

Le modificazioni post traduzionali (PTM - Post-Translational Modification in inglese) sono modifiche chimiche delle proteine seguenti alla loro traduzione. Le PTM possono estendere le funzioni delle proteine aggiungendo altri gruppi di funzioni biochimiche come gruppi acetici, gruppi fosfati, vari lipidi e glucidi, cambiando la natura chimica di un amminoacido o modificandone la struttura, come la formazione di ponti disolfuri. Tra le modificazioni post traduzionali la più studiata è la *fosforilazione*.

La fosforilazione è una reazione chimica che consiste nell'addizione di un gruppo fosfato ad una proteina o ad un'altra molecola. Tale reazione ha una frequenza molto elevata in biochimica: gli enzimi che solitamente catalizzano le fosforilazioni sono le chinasi.

Negli organismi eucarioti, la fosforilazione delle proteine è probabilmente il meccanismo di regolazione più importante. Molti enzimi e recettori vengono accesi e spenti attraverso eventi di fosforilazione e defosforilazione, attraverso l'azione specifica delle chinasi e delle fosfatasi (proteine deputate alla rimozione di un gruppo fosfato dalla proteina).

L'aggiunta di un fosfato (PO4) ad una catena laterale polare (come quella degli amminoacidi serina, treonina o tirosina) potrebbe sembrare un evento ininfluente per una grossa proteina. In realtà, basta una singola fosforilazione per convertire una proteina idrofobica ed apolare in una conformazione idrofilica ed estremamente polare.

La fosforilazione di una proteina, come accennato, può avere luogo su diversi residui amminoacidici. La fosforilazione della serina è la più comune, seguita da quella su treonina. La fosforilazione sulla tirosina è invece relativamente rara. In ogni caso, sebbene tali fosforilazioni siano rarissime, sono comunque le più studiate, soprattutto grazie alla facilità di purificazione che presentano e grazie ad anticorpi molto efficaci. Nei procarioti può avvenire anche la fosforilazione della istidina e dell'aspartato.

1.1.3 I database di fosforilazione attuali

Le informazioni sui processi di fosforilazione sono disseminate in una moltitudine di articoli scientifici. Alcuni database sono nati con l'intento di raccogliere tutti i processi di fosforilazione conosciuti in un unico database, rendendone più facile l'accesso da parte di tutta la comunità scientifica. I *dataset* attualmente esistenti sono importanti fonti di informazione, ma soffrono della mancanza di formalizzazione. Un esempio su tutti è la mancata distinzione, nella maggior parte dei dataset tra chinase e gruppo di chinasi.

1.1.4 Obiettivi del progetto

- I dataset di partenza non sono formalizzati e contengono dati eterogenei. *Formalizzare* e *disambiguare* queste informazioni migliora l'utilizzabilità dei dati stessi da parte di sistemi informatici.

- i dataset di partenza a volte sono contraddittori o registrano diversi siti di fosforilazione in posizioni diverse. Quindi uno degli obiettivi attuali del progetto consiste nel riuscire ad *identificare* queste *anomalie* ed evitare di includerle nel nuovo database. In un secondo momento sarà necessario elaborarle e risolvere il problema (si veda la sezione 6.3 sugli shift delle sequenze);

- il terzo obiettivo è quello di creare un sistema per estrarre automaticamente siti di fosforilazione in grandi quantità, ad esempio per *creare "fosforilomi"*, ovvero reti di fosforilazioni, con le quali capire meglio certi meccanismi cellulari. Ad esempio, i pathway di signalling sono spesso catene di fosforilazione (un esempio notevole è il pathway dell'insulina). I normali sistemi in generale permettono di estrarre solo 1 dato alla volta, oppure richiedono la scrittura di parsers ad hoc. L'uso di un database relazionale per formalizzare le informazioni ne semplifica l'accesso.

1.2 Lavoro svolto e risultati ottenuti

In questa sezione viene presentato il lavoro svolto nella tesi con i relativi risultati ottenuti. Come si vedrà meglio nel seguito (Capitolo 4) il progetto software è stato diviso in tre moduli: *Parser*, *Mapper* e *Populator*. Tali moduli rappresentano il risultato finale di 3 distinte fasi di lavoro, che vengono ora presentate.

1.2.1 Parsing dei dati grezzi

Inizialmente sono stati analizzati i quattro file con i dati dei database scelti per l'integrazione. Si sono cercate le caratteristiche particolari di ciascun file (le convenzioni utilizzate per memorizzare le informazioni, il tipo di informazioni presenti) e si è stabilito cosa fosse necessario estrarre e cosa fosse, invece, superfluo. Una volta deciso come procedere è stato scritto in Java il modulo *Parser* con lo scopo di elaborare le informazioni presenti nelle sorgenti dati e produrre dei nuovi file, più strutturati (si veda la Sezione 4.3 sul parser per maggiori dettagli).

1.2.2 Mapping degli ID dei quattro database

Successivamente sono stati analizzati in dettaglio i vari codici identificativi (ID) presenti nei quattro file che, di volta in volta, possono identificare o una proteina o un sito di fosforilazione (o anche altri tipi di informazioni come le strutture PDB o le pubblicazioni PubMed). Si sono quindi cercati dei codici comuni tra i vari database in modo da poter correlare tra loro i dati. Si è quindi creato un nuovo codice identificativo cui, attraverso il modulo *Mapper*, sono stati opportunamente collegati gli ID dei vari database, dove possibile. Come si vedrà meglio nella Sezione 4.4 sul mapper nessuno dei database è stato privilegiato rispetto agli altri.

1.2.3 Creazione e popolamento del database

La terza fase di sviluppo del progetto ha riguardato la progettazione e implementazione del nuovo database destinato a contenere le informazioni integrate. Si è quindi progettato lo schema ER, tradotto poi in relazionale e infine implementato in MySQL. È stato creato un terzo modulo software, il *Populator*, con lo scopo di creare automaticamente il database leggendone il codice sorgente da un file di testo. Questo terzo modulo si occupa anche del popolamento delle tabelle del nuovo database con le informazioni unificate estratte ed elaborate dalle sorgenti dati. Per maggiori dettagli sul modulo populator si rimanda alla Sezione 4.5.

1.2.4 Altri risultati ottenuti

Alcuni altri importanti risultati riguardano il progetto nel suo complesso, e non si riferiscono ad una specifica fase di sviluppo:

- i tempi di esecuzione sono stati *drasticamente* migliorati utilizzando estensivamente le strutture dati `HashSet` ed `HashMap` di Java (si veda anche la sezione 4.7);
- sono state eliminate dal nuovo database le informazioni sui siti di fosforilazione che sono risultate contradditorie tra i vari database (cio è dovuto a problemi di shifting delle sequenze primarie dei vari database). Per maggiori dettagli si rimanda alla Sezione 6.3;
- sono state distinte, in tutti i casi dove era possibile, le chinasi dai gruppi di chinasi. Dove ciò non è stato possibile si sono salvate le informazioni parziali in alcuni file di supporto che potranno essere utilizzati in futuro per risolvere le ambiguità, si veda la Sezione 6.2;

- sono state formalizzate molte informazioni che nei file originali erano presenti in formati ambigui, soprattutto per quanto riguarda le chinasi. Ad esempio molte volte la chinase era specificata solo da un "by autocatalysis", mentre ora possiede un proprio ID di riferimento;
- infine, grazie al nuovo database relazionale introdotto, è ora possibile eseguire query più o meno complesse, come quelle esemplificate nella Sezione 3.2.

1.3 Breve panoramica di questa tesi

Questa tesi si articola in sei capitoli.

- *Capitolo 1*: il primo capitolo contiene, oltre ad una panoramica sulla tesi stessa, una presentazione del problema affrontato e di quali obiettivi si sono voluti raggiungere;
- *capitolo 2*: in questo capitolo vengono presentate le caratteristiche dei quattro database scelti per l'integrazione nel nuovo database. Vengono analizzate le caratteristiche peculiari di ciascuna sorgente dati ed è spiegato quali informazioni vengono estratte di volta in volta, infatti non tutte le informazioni sono utili;
- *capitolo 3*: nel terzo capitolo viene illustrato il nuovo database. Prima ne vengono presentati gli schemi ER e relazionale con una spiegazione approfondita dell'uso delle varie entità e associazioni presenti. Vengono poi fornite alcune brevi considerazioni sul passaggio da schema ER a relazionale (una trattazione approfondita risulterebbe ridondante in quanto le informazioni necessarie sono esposte presentando lo schema ER). Infine viene svolta un'analisi sulla normalizzazione dello schema relazionale, una proprietà richiesta ad ogni buon database.

  Alla fine del capitolo 3 vengono anche forniti alcuni esempi di query realizzabili sul nuovo database per evidenziarne meglio le potenzialità.
- *capitolo 4*: il quarto capitolo riguarda il software Java sviluppato in questa tesi. Inizialmente vengono fornite alcune considerazioni sui vantaggi derivanti da una programmazione modulare che distingua tra interfacce e implementazioni e viene spiegata la suddivisione del programma in 3 moduli distinti. Successivamente viene illustrata la metodologia mista *Bottom-Up - Top-Down* che si è adottata nella realizzazione del software con le relative motivazioni e i vantaggi / svantaggi derivati da questa scelta.

Nella parte centrale del quarto capitolo vengono analizzate in dettaglio le interfacce e le classi che compongono i tre moduli del programma, con una spiegazione del funzionamento dei vari metodi implementati. Queste sezioni saranno utili sia a chi utilizzerà il software, sia a chi dovrà modificarne i metodi.

Infine viene spiegato come il software si occupi di creare automaticamente il nuovo database leggendone la struttura da un file di configurazione e vengono fornite alcune considerazioni sulle prestazioni complessive del software e sui problemi riscontrati in fase di sviluppo dello stesso;

- *capitolo* 5: questo capitolo tratta della documentazione fornita a corredo del software. Viene spiegata la scelta di produrre la documentazione in Javadoc e vengono forniti alcuni esempi della stessa. Successivamente sono presenti alcune considerazioni sulla riutilizzabilità del codice e sulla sua estendibilità, nonchè sulla necessità di eseguire periodicamente il programma, ogni volta che vengono rilasciate nuove versioni delle sorgenti dati. Infine vengono analizzati i due file di configurazione principali del programma: *config.cfg* e *tables.cfg*;

- *capitolo* 6: il sesto capitolo riguarda gli sviluppi futuri del progetto, a partire da quanto creato in questa tesi. Gli argomenti trattati sono:
  - la creazione di un *front end* web con cui interagire con il database attraverso una comoda interfaccia;
  - i problemi attualmente irrisolti relativi al parsing delle chinasi e ai collegamenti tra chinasi e gruppi di chinasi;
  - il problema dello shift delle sequenze primarie delle proteine nei diversi database, che, allo stato attuale, viene solo identificato;
  - alcuni problemi e/o migliorie minori.

Capitolo 2

Analisi dei database di fosforilazione.

2.1 Caratteristiche dei quattro database sorgenti

In questo capitolo vengono presentate le caratteristiche essenziali delle quattro sorgenti dati scelte per l'integrazione nel nuovo database: si tratta dei database *UniProt*, *PTM*, *ELM* ed *HPRD*. Il software sviluppato in questa tesi crea il nuovo database integrando tutte le informazioni utili dai quattro database e formalizzandole, ove possibile. La formalizzazione dei dati non è sempre possibile per quanto riguarda le chinasi perchè i file sorgenti usano una moltitudine di modi diversi di memorizzare le informazioni e in molti casi non è stato possibile estrarre automaticamente i dati corretti. Si rende quindi necessaria una futura fase di parsing più approfondita delle chinasi (si veda la sezione 6.2 relativa agli sviluppi futuri sulle chinasi).

Nelle sezioni seguenti vengono analizzati i formati dei quattro file di testo evidenziandone le particolarità specifiche. Inoltre per ogni database vengono specificate le informazioni estratte e riportate nel nuovo database.

2.1.1 Il database UniProt

UniProt contiene tutte le proteine, comprese quelle che non presentano siti di fosforilazione, a differenza degli altri database che memorizzano solo le proteine interessate alla fosforilazione. Di ogni proteina UniProt memorizza la sequenza primaria e l'ID della specie, informazioni che in genere non sono presenti negli altri database. D'altra parte alcuni siti presenti in altri database possono non essere indicati da UniProt. Nel nuovo database, oltre ai siti di fosforilazione, vengono mantenute anche le informazioni sulla specie e sulla sequenza primaria.

UniProt salva i dati in un file di testo nel quale un singolo record è disposto su righe multiple (a differenza degli altri tre database utilizzati). Ogni record inizia con i caratteri `ID` e termina con \\.
Ciascuna riga del record può contenere informazioni di diverso tipo, per distinguere le righe viene utilizzato un codice alfabetico di due caratteri. Questi due caratteri sono posti all'inizio della riga e permettono di capirne il contenuto. Di seguito si riporta una descrizione sommaria dei possibili codici utilizzati.

- **ID**: è la riga, univoca, contenente l'ID UniProt della proteina, il suo stato (revisionata o meno) e la sua dimensione (numero di aminoacidi);
- **AC**: è la riga contenente tutti gli AC (*ACcession number*) collegati alla proteina. Ci possono essere più righe AC. Un AC è un codice identificativo univoco per una sequenza di DNA o una proteina usato da molti database;
- **DT**: le tre righe DT indicano le date di creazione e ultima modifica del record;
- **DE**: contiene informazioni di descrizione della proteina;
- **GN**: è la riga contenente il nome del gene e/o altri dati sul gene;
- **OS**: questa riga contiene i nomi latino e inglese della specie di appartenenza;
- **OG**:
- **OC**: contiene la classificazione tassonomica dell'organismo;
- **OX**: contiene informazioni sulla specie di appartenenza della proteina, in particolare contiene l'ID tassonomico NCBI;

- **OH**: è una riga opzionale presente solo per i virus e contiene gli ID tassonomici delle specie ospite;
- **RN**: tutte le righe che iniziano con R indicano informazioni di riferimento bibliografico, ciascun tipo di riga R contiene informazioni diverse. RN indica il numero progressivo del riferimento bibliografico per il record;
- **RP**: serve a identificare il tipo di informazioni presenti nella citazione;
- **RC**: queste righe contengono commenti riguardo alla citazione;
- **RX**: contiene gli ID della citazione bibliografica usati da altri database bibliografici, in particolare contiene gli ID PubMed;
- **RG**: contiene i nomi dei consorzi associati alla citazione;
- **RA**: contiene gli autori della citazione;
- **RT**: contiene il titolo della citazione;
- **RL**: contiene informazioni utilizzabili per trovare la pubblicazione che viene citata;
- **CC**: le righe CC contengono commenti testuali del record, suddivisi in "argomenti" predefiniti;
- **DR**: contiene ID di riferimento ad altri database, come ad esempio l'ID PDB;
- **PE**: indica il grado di certezza di esistenza della proteina;
- **KW**: contiene parole chiave utilizzabili per indicizzare il record;
- **FT**: le righe FT contengono, tra le altre cose, le informazioni sui siti e i tipi di fosforilazione;
- **SQ**: la riga SQ indica l'inizio della sequenza primaria, che sarà memorizzata su più righe, di cui solo la prima iniza con SQ.

Non tutte le righe sono sempre presenti per tutti i record e alcune possono essere presenti più volte (sia in numero fisso (DT) che in numero variabile (ad esempio CC)), per un approfondimento sui codici e i possibili contenuti delle varie righe si rimanda al manuale online UniProt. In questa tesi vengono trattati solo gli aspetti di interesse per il progetto.
I dati che, per ogni record, vengono estratti dal file UniProt in fase di parsing sono:

- ID della proteina, stato, e numero di aminoacidi dalla riga `ID`;
- tutti gli AC presenti nelle righe `AC` (sono separati da ";");
- l'ID tassonomico della specie dalla riga `OX`;
- gli ID PubMed presenti nelle righe `RX`;
- i siti e le informazioni sulla fosforilazione presenti nelle righe `FT`. Le informazioni di fosforilazione sono presenti solo sulle righe `FT` di tipo `MOD_RES`;
- la sequenza primaria della proteina presente nelle righe `SQ` (la prima riga SQ contiene solo un'intestazione);

Data la complessità della struttura dei record UniProt il parsing viene gestito utilizzando un'intera classe di gestione dei record, cosa che per gli altri tre file non è stata necessaria (per maggiori dettagli si veda l'interfaccia `UniProtRecord` - sezione 4.3.5).

Per quanto riguarda le chinasi il file UniProt contiene sia informazioni certe (le proteine che si auto-fosforilano) sia dati per i quali non si può distinguere tra chinase e gruppo di chinasi. Inoltre alcune informazioni sulle chinasi sono in un formato ambiguo e vengono quindi inserite in un file di dati sconosciuti. Tra i dati di fosforilazione è presente anche il grado di ipotesi, non ancora provata attraverso sperimentazione, relativa al sito fosforilato (per similitudine, potenziale o probabile).
Il parsing delle chinasi UniProt viene fatto in una seconda fase, separata dalla fase di parsing generale del file a causa proprio dell'eterogeneità dei dati presenti.
Come già accennato le uniche chinasi certe in UniPort sono quelle proteine che si auto-fosforilano, quelle, cioè, che hanno la dicitura "autocatalysis" nella descrizione. In tutti gli altri casi o non si può avere la certezza che non siano gruppi o addirittura non si tratta proprio di nomi di chinasi e in quest'ultimo caso i dati vengono inseriti nel file "Uni_unknown.txt" che contiene ad esempio:

```
Phosphotyrosine; alternate (By similarity).
Phosphoserine; in alleles A and C.
Phosphothreonine; in form 6-P and form 7- P.
Phosphoserine; in isoform HMG-I and isoform HMG-Y.
Phosphoserine; by RAF; alternate.
```

2.1.2 Il database PTM

I dati del database PTM sono contenuti in un file di testo dove i record sono scritti su una singola riga e i campi sono separati da tabulazioni.
PTM prende i suoi dati da tre diversi database: **Swiss-Prot 53**, **Phospho.ELM 6** e **OglycBase 6**.
I campi presenti nei record del file sono:

- il primo dato è l'*ID* della proteina, in formato UniProt;
- la seconda informazione è la *posizione* del sito di fosforilazione;
- il terzo campo è il *tipo* di fosforilazione per il sito in questione;
- il quarto dato sono le *pubblicazioni* di riferimento per il particolare sito;
- infine viene memorizzato il database *sorgente* da cui sono stati presi i dati.

Per la creazione del nuovo database solo i primi quattro campi sono di interesse.
L'ID della proteina può essere ripetuto più volte perchè i record del file si riferiscono a singoli siti di fosforilazione piuttosto che a proteine.
Non tutti i record del file sono utili alla creazione del nuovo database, in particolare sono stati considerati solo quelli relativi a fosforilazione (per i quali il terzo campo del record contiene la parola "phospho"), per farlo è bastato analizzare i valori del terzo campo andando quindi a escludere record del tipo:

```
RFP_DISSP 66 2-iminomethyl-5-imidazolinone (Gln-Gly). Swiss-Prot 53.0
QADG_PARDE 41 3-cysteinyl-aspartic acid (Cys-Asp). Swiss-Prot 53.0
YNW8_YEAST 118 N-linked (GlcNAc...) (Potential). Swiss-Prot 53.0
```

Il file PTM contiene poche informazioni riguardanti le pubblicazioni PubMed, ma quando questi dati sono presenti possono avere valori multipli, separati da un carattere ";". È stato necessario tenere in considerazione anche questa particolarità per non perdere informazioni nella fase di parsing.
Le informazioni riguardanti le chinasi, nel file PTM, sono contenute nel terzo campo, quello relativo al tipo di fosforilazione, e possono avere forme molto diverse tra loro, alcuni esempi:

```
Phosphotyrosine (by autocatalysis) (Bysimilarity).
Phosphoserine (by autocatalysis andMAP4K1) (By similarity).
Phosphotyrosine (by autocatalysis)(Probable).
Phosphoserine (by PKA).
Phosphothreonine; by PDPK1 and PKB/AKT1;in vitro.
Phosphoserine (MAPK1;MAPK3;)
Phosphoserine (PKA_group)
Phosphothreonine (Potential).
Phosphoserine (CaM-KII_grou)
```

Data l'evidente eterogeneità nel rappresentare le informazioni si è resa necessaria una complessa fase di parsing delle stesse. Per tale motivo si è scelto di dividere, a livello di codice, i metodi usati per il parsing generale del file e quelli per il parsing specifico delle chinasi, in modo da aumentare la leggibilità del codice stesso.
La dicitura tra parentesi presente in alcune righe può assumere solo i tre valori *by similarity*, *probable*, e *potential* ed indica il tipo di conoscenza che esiste in merito alla specifica fosforilazione, è un'informazione importante che viene estratta in fase di parsing e salvata nel nuovo database.
Infine quando in una riga è presente la voce *autocatalysis* significa che la proteina fosforila se stessa: per il database PTM questo è l'unico caso in cui si può essere certi che la chinasi non sia un gruppo e si può quindi inserirla nella tabella delle chinasi.
Invece per quanto riguarda i gruppi bisogna verificare se nel nome è presente una stringa del tipo _*gro*, _*grou* o _*group* infatti non è sempre presente la parola *group* completa (probabilmente perché i dati sono danneggiati).

2.1.3 Il database ELM

I dati del database ELMM sono contenuti in un file di testo in cui i record sono scritti su una singola riga e i campi sono separati da tabulazioni.
Come per PTM anche in ELM esiste un record (una riga) per ogni sito di fosforilazione, ma a differenza del database PTM, in questo file è presente una riga di intestazione per specificare il contenuto delle colonne, ovvero dei campi dei record:

- **pELM_ID** è l'ID in formato ELM del sito di fosforilazione, ovvero l'ID ELM non fa riferimento alle proteine ma ai siti, è quindi univoco per ogni record;
- **acc** è l'accession number (AC) e identifica univocamente le proteine, quindi può essere ripetuto più volte se, come spesso accade, per una proteina sono presenti più siti;
- **sequence** è la sequenza primaria di aminoacidi di cui è composta la proteina, viene ripetuta per ogni sito;
- **position** indica la posizione del sito di fosforilazione;
- **code** è il tipo di fosforilazione, codificato con una singola lettera (*S*, *T*, *H*, *Y*, *C*, *R*);
- **pmids** sono gli ID PubMed di riferimento per il sito presente nel record, possono essere multipli e separati da ";";
- **kinases** sono i dati relativi alle chinasi (o gruppi di chinasi) per il sito corrente;
- **source** indica il tipo di esperimento utilizzato per ottenere le informazioni;
- **species** è il nome della specie di appartenenza della proteina, scritto in Latino;
- **entry_date** è la data di inserimento del record nel database.

Tutti i campi sono utili per il nuovo database, e vengono estratti in fase di parsing, tranne l'ultimo: infatti non è di alcuna utilità, nè attuale nè futura, conservare l'indicazione della data di inserimento in ELM.
Tutte le righe (i record) del file ELM sono utili e vengono estratte nel file elaborato. Alcuni campi dei record possono avere valori multipli che vanno analizzati ed eventualmente salvati (in fase di popolamento del database bisognerà evitare le eventuali duplicazioni di dati). In particolare ogni record ha sempre un solo ID ed un solo AC, così come è unica la sequenza primaria. Gli altri tre campi a valore unico sono posizione e tipo del sito di fosforilazione e la specie della proteina. Invece per quanto riguarda PubMed, chinasi e tipo di esperimento i dati possono essere multipli, nel qual caso sono separati da un ";", ovvero ELM usa delle convenzioni

piuttosto uniformi e quindi risulta molto più semplice eseguirne il parsing rispetto al file PTM.

Il file ELM contiene sia indicazioni di chinasi che di gruppi di chinasi. Per i gruppi il nome è sempre nella forma "**X_group**" ed è quindi immediato distinguerlo, mentre per quanto riguarda le singole chinasi, allo stato attuale, non è implementato un modo per essere certi non si tratti in realtà di un gruppo, quindi dove non è presente la dicitura "_group" la chinasi viene inserita nel file con i dati incerti (si veda la sezione 6.2 relativa agli sviluppi futuri sulle chinasi).
Il file ELM non contiene indicazioni di tipo "by similarity", "potential" o "probable" per quanto riguarda le sue chinasi.

2.1.4 Il database HPRD

I dati del database HPRD sono contenuti in un file di testo, un record per ogni riga, come accade per ELM e PTM. Anche nel file HPRD i campi dei record sono separati da tabulazioni. Ogni record del file contiene le informazioni relative ad un sito di fosforilazione.
I campi presenti nei record del file sono:

- il primo dato è l'*ID* HPRD della proteina. Si tratta di un ID numerico che può essere ripetuto su più righe;
- la seconda informazione è il *GeneSymbol* della proteina in questione. Analogamente all'ID HPRD può essere ripetuto su più righe;
- il terzo campo è l'*isoform* che identifica la sequenza primaria di riferimento per il sito. Una singola proteina può avere diversi isoform, ovvero diverse rappresentazioni della sequenza primaria;
- il quarto dato è l'ID *RefSeq*;
- il quinto campo indica la *posizione* del sito di fosforilazione;
- la sesta informazione è il *tipo* di fosforilazione, codificata da una singola lettera;
- il settimo dato nel file è il *nome della chinase*, se l'informazione non è presente viene sostituita da un "-";
- l'ottavo campo è l'*ID HPRD della chinase*, come per il campo precedente, se l'informazione non è presente viene sostituita da un "-";
- la nona informazione è il *tipo di modificazione* memorizzata nel record, vengono estratte solo le fosforilazioni, ma il database HPRD contiene anche altre informazioni, ad esempio metilazioni o acetilazioni;
- il decimo campo rappresenta il tipo di esperimento;
- infine vengono memorizzati gli *ID PubMed* di riferimento, separati da "," in caso di valori multipli.

Allo stato attuale non tutti i campi del file solo utili, in particolare vengono esclusi i campi del RefSeq e del tipo di modificazione. Ovvero il RefSeq non viene proprio estratto mentre il tipo di modificazione viene usato per decidere quali record tenere (quelli riguardanti la fosforilazione) ma non viene salvato in quanto è sufficiente il tipo di fosforilazione (sesto campo del record).
L'unico campo dei record HPRD che può avere valori multipli è quello relativo agli ID PubMed, tuttavia alcune posizioni dei siti di fosforilazione sono memorizzate

nella forma `xxx;-` (dove `xxx` è un numero) ed è quindi necessario togliere i caratteri ";-".

Il file HPRD contiene solo informazioni sulle chinasi, non sui gruppi, ma tutti i dati presenti sono certi in quanto è sempre presente l'ID della chinase in formato HPRD. Per questo motivo, come indicato anche nell'interfaccia `Parser` - sezione 4.3.6, non è necessaria una seconda fase di parsing riservata alle chinasi come invece avviene per gli altri tre database.
Il file HPRD, come il file ELM, non contiene indicazioni di tipo "by similarity", "potential" o "probable" per quanto riguarda le sue chinasi.

Infine il database HPRD fornisce un secondo file che viene utilizzato dal programma, il file "HPRD_ID_MAPPINGS.txt". Anche questo file contiene un record per ogni riga, con i campi separati da tabulazioni, tuttavia gli ID non vengono ripetuti in quanto i record di questo file rappresentano delle proteine anzichè dei siti. La funzione del file è quella di fornire una mappa tra gli ID HPRD ed altri tipi di ID, in particolare tra questi ID ci sono gli *Accession Numbers* (AC) con i quali è possibile stabilire un collegamento tra una proteina HPRD ed una UniProt (o una di un qualsiasi altro database che utilizzi gli AC). Gli altri ID mappati nel file non sono di interesse per questo progetto.

Per alcuni record possono esserci più AC collegati allo stesso ID, in questo caso gli AC sono separati da una ",". Per creare un collegamento con una proteina di un altro database è sufficiente che almeno uno degli AC venga trovato.

I quattro database non solo usano convenzioni diverse per memorizzare i dati nei file, ma non memorizzano neanche tutti lo stesso tipo di informazioni. Alcuni sono più completi di altri e in alcuni casi due o più database contengono informazioni contrastanti riguardo il medesimo soggetto (un sito di fosforilazione).

Capitolo 3

Il nuovo database *PhosphoDB*.

In questo capitolo vengono illustrate le caratteristiche del nuovo database *PhosphoDB* sviluppato in questa tesi presentandone gli schemi ER e relazionale. Viene analizzata la conformità alle forme normali richiesta per un buon database e vengono fornite alcune query di esempio.
Come si vedrà nel seguito, il database presenta una forte simmetria centrata sulle entità di tipo *proteina* e *sito*, ciò rende molto semplice l'espansione futura del database stesso per includere qualche nuova sorgente di dati di fosforilazione.

3.1 Caratteristiche del nuovo database

Lo scopo del database è di unificare e formalizzare le informazioni presenti nei quattro database scelti, per questo viene creata un'entità *Protein* che rappresenta una proteina del nuovo database, ma vengono mantenute anche tutte le informazioni sulle proteine dei quattro database di partenza. Questo viene fatto anche per i siti di fosforilazione e per ogni altro dato, in questo modo è sempre possibile risalire alla fonte delle informazioni unificate presenti nel database. Per esempio è possibile cercare tutte le proteine della specie "Human" che provengono sia da UniProt che da ELM.

3.1.1 Schema ER del nuovo database

Durante la fase di analisi del problema sono state identificate sette entità base:

- *proteina*, rappresenta una proteina e contiene informazioni quali la sequenza primaria e la specie di appartenenza. Ci sono cinque diverse entità proteina, una per ogni database sorgente più quella del nuovo database. Ciascuna proteina dei database sorgenti è collegata alla nuova proteina in quanto ne costituisce una sorgente di dati;

- *sito*, rappresenta un sito di fosforilazione di una determinata proteina. Contiene, quindi, oltre ad un riferimento alla proteina, la posizione del sito e il tipo di fosforilazione, più altri dati specifici del database. Infatti anche in questo caso esistono cinque diverse entità di tipo sito;

- *chinase*, sebbene le chinasi siano di fatto delle proteine si è scelto di distinguere le due entità per ragioni pratiche. Infatti l'entità chinase è unica ed è collegata a molte entità sito, non avrebbe senso distinguere tra chinase UniProt e chinase PTM, per esempio;

- *gruppo di chinasi*, anche questa entità, per ragioni analoghe a quelle relative all'entità chinase, è unica per tutto il database;

- *specie*, altra entità unica per tutto il database, mantiene le informazioni sulle specie cui fanno riferimento le proteine presenti nel database;

- *dati PDB*, questa entità contiene le informazioni relative alla sequenza terziaria per alcune proteine;

- *pubblicazione PubMed*, è semplicemente l'ID PubMed di riferimento di una pubblicazione relativa ad un sito di fosforilazione o ad una proteina (nel caso di UniProt).

Di seguito viene riportato lo schema ER del nuovo database: gli attributi sottolineati costituiscono la chiave primaria, se sono anche in corsivo costituiscono una chiave parziale per le entità deboli. Le relazioni identificanti e le entità deboli sono rappresentate, rispettivamente, da rombi e da rettangoli con bordo doppio. Gli attributi delle entità/relazioni sono indicati all'interno di ellissi. Poichè lo schema ER è uno schema concettuale non vengono indicate le chiavi esterne.

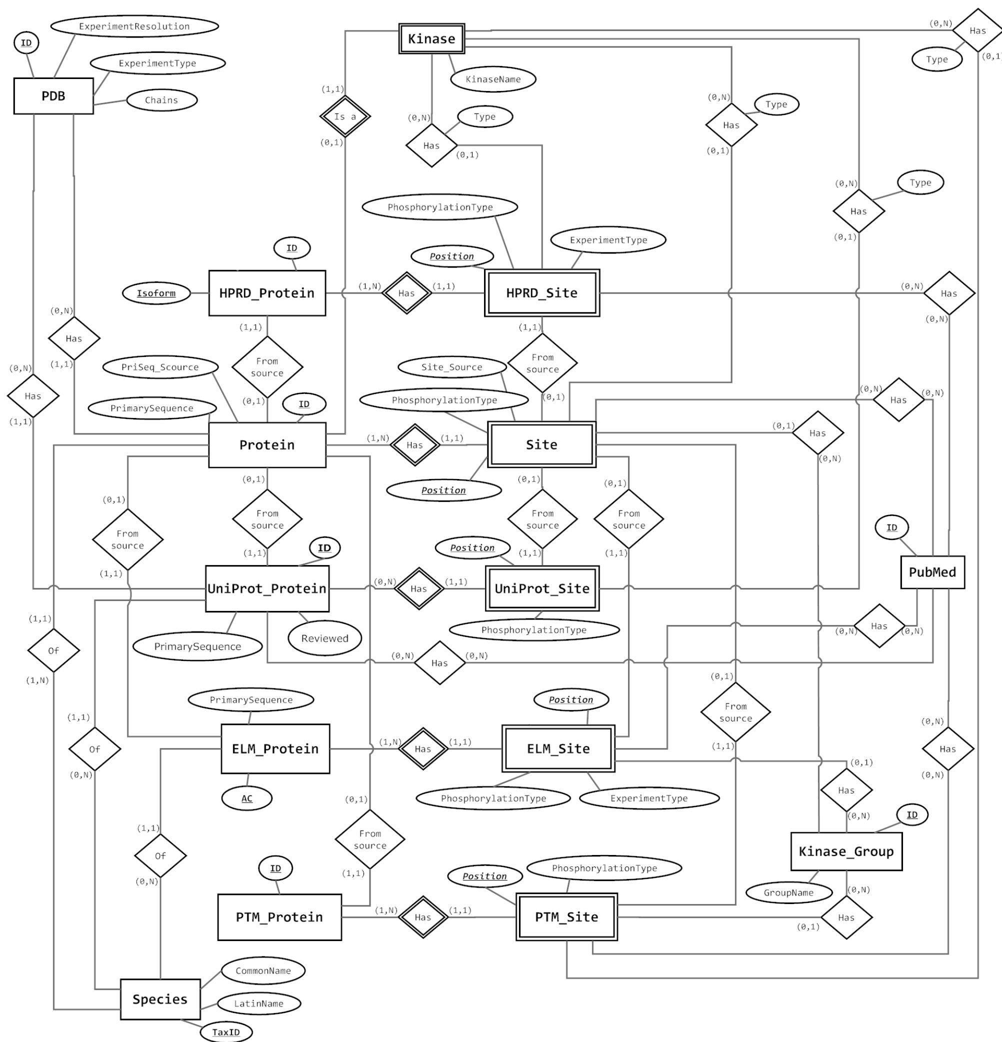

Figura 3.1: Schema ER del nuovo database.

Analisi delle entità presenti

Alcune entità come le proteine o i siti possono avere attributi comuni, ma anche caratteristiche specifiche dipendenti dal database di origine. In questa sezione sono riportate, in dettaglio, le caratteristiche di tutte le entità coinvolte.

- *Protein*: rappresenta una proteina risultante dall'unificazione dei dati dei quattro database sorgenti. I suoi attributi sono:
  - *ID* è un numero progressivo creato dal programma in fase di mapping dei dati e identifica univocamente le proteine;
  - *PrimarySequence* contiene la sequenza primaria della proteina se tale sequenza è uguale in tutti i database che la memorizzano, in caso contrario nessuna sequenza viene memorizzata;
  - *PriSeq_Source* è un campo testuale che memorizza i nomi dei database sorgenti nei quali è stato possibile trovare l'esatta sequenza primaria memorizzata per la proteina.
- *UniProt_Protein*: rappresenta una proteina contenuta nel database UniProt. I suoi attributi sono:
  - *ID* è l'ID della proteina usato dal database UniProt. Costituisce la chiave primaria dell'entità;
  - *Reviewed* indica lo stato della proteina UniProt;
  - *PrimarySequence* è la sequenza primaria della proteina contenuta in UniProt.
- *HPRD_Protein*: rappresenta una proteina contenuta nel database HPRD. I suoi attributi sono:
  - *ID* è l'ID della proteina usato dal database HPRD;
  - *Isoform* indica quale particolare isoform della sequenza primaria è usato. Costituisce, insieme all'*ID*, la chiave primaria dell'entità;
  - *GeneSymbol* è un campo testuale che memorizza il nome del gene presente nel database HPRD.
- *PTM_Protein*: rappresenta una proteina contenuta nel database PTM. I suoi attributi sono:
  - *ID* è l'ID della proteina usato dal database PTM, formalmente è lo stesso usato da UniProt e costituisce la chiave primaria dell'entità.
- *ELM_Protein*: rappresenta una proteina contenuta nel database ELM. I suoi attributi sono:

- *AC* è l'ID della proteina usato dal database ELM, il quale utilizza direttamente gli acession numbers. Costituisce la chiave primaria dell'entità;
- *PrimarySequence* è la sequenza primaria della proteina contenuta in ELM.

- *Site*: rappresenta un sito di fosforilazione risultante dall'unificazione dei dati dei quattro database sorgenti. I suoi attributi sono:
  - *Position* è la posizione del sito nella proteina e costituisce la chiave parziale dell'entità;
  - *PhosphorylationType* è il tipo di fosforilazione, rappresentata da una singola lettera;
  - *Site_Source* è un campo testuale che memorizza i nomi dei database sorgenti nei quali è stato possibile trovare il sito di fosforilazione.
- *UniProt_Site*: rappresenta un sito di fosforilazione presente nel database UniProt. I suoi attributi sono:
  - *Position* è la posizione del sito nella proteina e costituisce la chiave parziale dell'entità;
  - *PhosphorylationType* è il tipo di fosforilazione, rappresentata da una singola lettera.
- *HPRD_Site*: rappresenta un sito di fosforilazione presente nel database HPRD. I suoi attributi sono:
  - *Position* è la posizione del sito nella proteina e costituisce la chiave parziale dell'entità;
  - *PhosphorylationType* è il tipo di fosforilazione, rappresentata da una singola lettera;
  - *ExperimentType* è un campo testale contenente tutti i tipi di esperimento utilizzati per ricavare le informazioni sul sito, come indicato nel database HPRD.
- *PTM_Site*: rappresenta un sito di fosforilazione presente nel database PTM. I suoi attributi sono:
  - *Position* è la posizione del sito nella proteina e costituisce la chiave parziale dell'entità;
  - *PhosphorylationType* è il tipo di fosforilazione, rappresentata da una singola lettera.

- *ELM_ Site*: rappresenta un sito di fosforilazione presente nel database ELM. I suoi attributi sono:
  - *Position* è la posizione del sito nella proteina e costituisce la chiave parziale dell'entità;
  - *PhosphorylationType* è il tipo di fosforilazione, rappresentata da una singola lettera;
  - *ExperimentType* è un campo testale contenente tutti i tipi di esperimento utilizzati per ricavare le informazioni sul sito, come indicato nel database ELM;
  - *ID* è l'ID numerico ELM che identifica il sito di fosforilazione.
- *Kinase*: rappresenta una chinase. I suoi attributi sono:
  - *KinaseName* il nome della chinase estratto dai database sorgenti. Una chinase è di fatto una proteina e quindi viene identificata solo mediante la relazione ISA.
- *Kinase_ Group*: rappresenta un gruppo di chinasi. I suoi attributi sono:
  - *ID* è un campo numerico generato dal programma che identifica i gruppi. Costituisce la chiave primaria dell'entità;
  - *GroupName* è il nome del gruppo estratto dai database sorgenti.
- *PDB*: rappresenta i dati PDB sulla sequenza terziaria delle proteine. I suoi attributi sono:
  - *ID* è l'ID trovato nel database UniProt, usato come chiave primaria dell'entità;
  - *ExperimentType* è il tipo di eseprimento usato per ottenere i dati;
  - *ExperimentResolution* indica la precisione dell'esperimento;
  - *Chains* è un campo testuale che indica le catene di aminoacidi.
- *Species*: rappresenta le specie cui fanno riferimento le proteine del database. I suoi attributi sono:
  - *TaxID* è l'ID tassonomico usato dal database NCBI per identificare le specie, viene usato come chiave primaria dell'entità;
  - *LatinName* è il nome scientifico della specie;
  - *CommonName* è il nome comune della specie;
- *PubMed*: rappresenta le pubblicazioni PubMed cui fanno riferimento i dati contenuti nel database. I suoi attributi sono:
  - *ID* è il codice identificativo delle pubblicazioni PubMed e costituisce anche la chiave primaria dell'entità;

Analisi delle associazioni presenti

Sebbene tra le entità siano presenti molte associazioni, queste si possono raggruppare in alcuni tipi ripetuti più volte:

- *associazioni proteina-sito*, esprimono l'appartenenza ad una proteina dei vari siti di fosforilazione memorizzati;
- *associazioni proteina-proteina*, esprimono il collegamento tra i dati sorgenti delle proteine e i dati unificati della nuova entità *Protein*;
- *associazioni sito-sito*, esprimono il collegamento tra i dati sorgenti dei siti di fosforilazione e i dati unificati della nuova entità *Site*;
- *associazioni sito/proteina-PubMed*, esprimono il collegamento tra i dati di un sito di fosforilazione o di una proteina (nel caso di UniProt) e gli ID delle pubblicazioni PubMed che ne parlano;
- *associazioni proteina-PDB*, esprimono il collegamento tra le proteine e gli ID delle sequenze terziarie che le rappresentano;
- *associazioni sito-chinase*, esprimono il collegamento tra i siti di fosforilazione e le chinasi che fosforilano tali siti;
- *associazioni sito-gruppo di chinasi*, esprimono il collegamento tra i siti di fosforilazione e i gruppi di chinasi che li fosforilano;
- *associazioni proteina-specie*, esprimono il collegamento tra una proteina e la specie dell'organismo cui appartiene;
- *associazioni ISA* esprime il fatto che una chinase in realtà è una proteina, attraverso questa associazione è possibile identificare le chinasi mediante gli ID dell'entità *Protein*.

Le associazioni di tipo sito-chinase contengono un attributo proprio dell'associazione chiamato *Type* nello schema. Questo attributo memorizza il dato sull'affidabilità della fosforilazione, quando presente. Può quindi assumere i valori "by similarity", "potential" e "probable", oltre che essere lasciato vuoto.

3.1.2 Schema logico relazionale del nuovo database

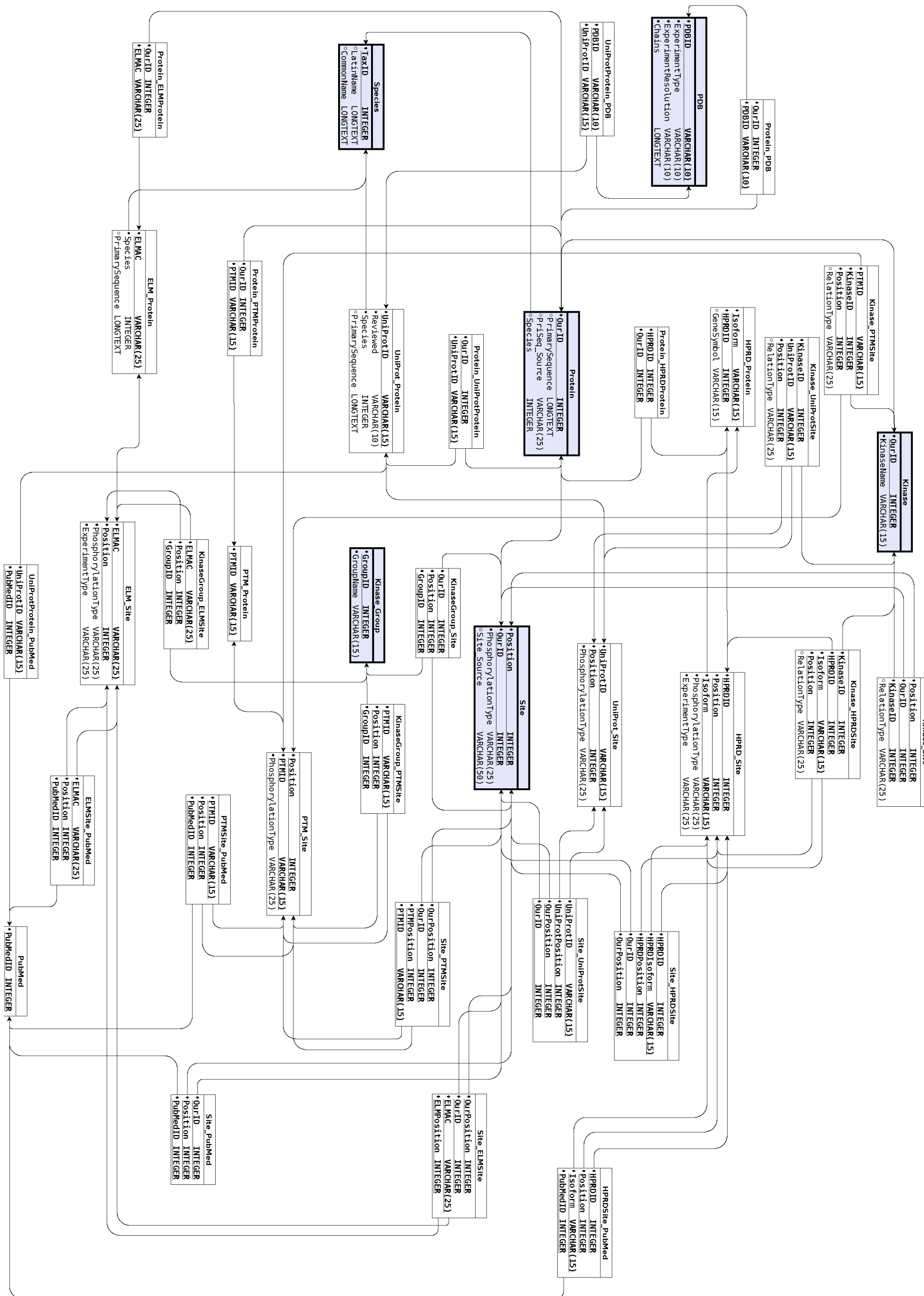

Figura 3.2: Schema logico-relazionale del nuovo database.

Tutte le entità presenti nello schema ER sono state trasformate nelle rispettive relazioni mantenendo tutti gli attributi indicati ed aggiungendo, quando necessario, alcuni attributi chiave esterna, soprattutto per quanto riguarda le entità deboli di tipo "sito di fosforilazione", le cui chiavi primarie utilizzano alcuni attributi della rispettiva proteina.

Per quanto riguarda le associazioni, invece, alcune non sono state tradotte con delle relazioni, in quanto esprimono un collegamento di tipo 1:N come ad esempio l'associazione tra le entità *Protein* e *Species* per la quale è bastato aggiungere una chiave esterna nella relazione *Protein* per collegare le due relazioni (più proteine possono appartenere alla stessa specie, la quale, però, è unica per una data proteina). In altri casi è stato necessario tradurre l'associazione con una nuova relazione in quanto il collegamento è di tipo molti-a-molti: ad esempio nel caso delle associazioni tra siti di fosforilazione e pubblicazioni PubMed. Nel seguito queste relazioni verranno indicate con "relazioni (o tabelle) di collegamento".

Le relazioni tra sito e chinasi, o gruppi di chinasi, potevano essere omesse aggiungedo tre campi alle relazioni di tipo sito:

- un campo chiave esterna per l'ID della chinase, eventualmente nullo;
- un campo chiave esterna per l'ID del gruppo di chinasi, eventualmente nullo;
- un campo testuale per il tipo di informazione (by similarity, potential, probable), eventualmente vuoto.

Si è preferito evitare questo approccio perchè per la maggior parte dei siti di fosforilazione non è presente l'informazione sulla chinasi, e/o sul gruppo, quindi si sarebbero introdotti molti valori `NULL` nelle tabelle, appesantendo inutilmente il database.

Per ragioni analoghe, e per permettere una facile estendibilità futura del database a nuove sorgenti dati, si sono introdotte le relazioni di collegamento proteina-proteina e sito-sito. Ad esempio nella relazione *Protein* si sarebbero potute aggiungere quattro chiavi esterne alle quattro tabelle delle proteine dei database sorgenti, introducendo però tantissimi valori `NULL`. Inoltre in questo modo bisognerebbe modificare anche la relazione *Protein* nel caso venisse introdotta una nuova sorgente dati. Invece con l'approccio scelto sarà sufficiente introdurre una nuova relazione di collegamento. Considerazioni analoghe valgono per la relazione *Site*.

3.1.3 Analisi sulle forme normali del nuovo database

Alcune definizioni utili nel seguito

- *Chiave candidata*: è un insieme di attributi di una relazione che consente di identificarne univocamente le tuple. In una relazione possono esserci più chiavi candidate, una di esse viene scelta come chiave primaria;

- *Attributo primo*: in uno schema di relazione, si dice primo un attributo che è parte di una qualche chiave candidata dello schema, mentre si dice non primo un attributo che non fa parte di alcuna chiave candidata dello schema;

- *1NF - Prima forma normale*: uno schema di relazione R è in prima forma normale (*1NF*) se il dominio di un attributo comprende solo valori atomici ed è un valore singolo del dominio dell'attributo stesso;

- *2NF - Seconda forma normale*: uno schema di relazione R è in seconda forma normale (*2NF*) se rispetta la prima forma normale e se ogni attributo non-primo A di R non è parzialmente dipendente da alcuna chiave di R;

- *3NF - Terza forma normale*: uno schema di relazione R è in terza forma normale (*3NF*) se rispetta la seconda forma normale e se per ogni dipendenza funzionale (non banale) $X \rightarrow Y$ definita su di esso, o X è una superchiave di R o Y è un attributo primo di R.

- *BCNF - Forma normale di Boyce e Codd*: uno schema di relazione R è in forma normale di Boyce e Codd (*BCNF*) se rispetta la terza forma normale e se per ogni dipendenza funzionale (non banale) $X \rightarrow Y$ definita su di essa, X è una superchiave di R.

Un buono schema di relazione dovrebbe rispettare almeno le prime tre forme normali (la forma normale di Boyce e Codd impone un vincolo un po' più stringente rispetto alla 3NF).

Prima forma normale

Si può facilmente verificare che lo schema proposto rispetta la prima forma normale, infatti gli ID PubMed collegati ai siti di fosforilazione potevano costituire l'unico caso di attributo multiplo, ma utilizzando le relazioni di collegamento la 1NF viene rispettata.

Esistono alcuni attributi che potrebbero essere visti come multivalore, per esempio il campo *PriSeq_Source* della relazione *Protein*, il campo *ExperimentType* della relazione *HPRD_Site* o altri simili ma di fatto le informazioni presenti in questi attributi sono delle semplici stringhe che non fanno riferimento ad altre tabelle e sono inserite solo per completezza. In pratica i valori inseriti in questi attributi, se multipli, vengono concatenati opportunamente dal programma in fase di parsing

e/o popolamento e inseriti come un singolo valore atomico. Predisporre ulteriori relazioni di collegamento per rendere questi campi atomici sarebbe un inutile appesantimento del database (servirebbero due tabelle per ogni campo: una con i singoli valori atomici, e una di collegamento).

Per esempio, per l'*ExperimentType* della relazione *ELM_Site* per quei siti che contengono sia l'esperimento *HTP* che l'esperimento *LTP* bisognerebbe creare una tabella con la lista dei possibili tipi di esperimento presenti in tutti i database (quindi *HTP* ed *LTP* da ELM, *in vitro* e *in vivo* da HPRD, ecc) e una tabella (relazione) di collegamento tra questa nuova tabella e il sito interessato. Come già detto, data la natura di queste informazioni, si è valutata superflua l'aggiunta queste tabelle.

Seconda forma normale

È immediato notare che la seconda forma normale è rispettata per tutte le relazioni dove la chiave primaria è composta da un solo attributo e per quelle in cui tutti gli attributi costituiscono la chiave primaria. Infatti nel primo caso gli attributi non-primi non possono dipendere parzialmente dalla chiave, essendo questa costituita di un solo attributo, mentre nel secondo caso tutti gli attributi sono primi.

Occorre verificare la seconda forma normale per le restanti relazioni. Per semplicità le relazioni in questione vengono raggruppate in quanto sono dello stesso tipo, ma riferite a database sorgenti diversi e quindi le considerazioni valide per una sono analoghe per le altre.

- Relazioni di tipo *sito di fosforilazione*: comprende le relazioni *Site*, *UniProt_Site*, *HPRD_Site*, *PTM_Site* ed *ELM_Site*. Per queste relazioni si può vedere che la seconda forma normale è rispettata perchè le informazioni relative al sito quali il tipo di fosforilazione e/o il tipo di esperimento condotto e, nel caso della relazione *Site*, la sorgente dei dati dipendono in modo completo sia dalla posizione del sito che dalla proteina di riferimento. Infatti proteine diverse possono avere fosforilazioni diverse nella "stessa" posizione;

- relazioni di tipo *chinase-sito*: comprende le relazioni *Kinase_Site*, *Kinase_UniProtSite*, *Kinase_HPRDSite* e *Kinase_PTMSite*. Per queste relazioni si può vedere che la seconda forma normale è rispettata perchè il tipo di relazione ("by similarity", "potential" o "probable") dipende sia dal sito (e quindi anche dalla proteina come visto sopra) che dalla specifica chinase che lo fosforila. È infatti possibile che un sito vegna fosforilato da diverse chinasi e il tipo di relazione dipende anche dalla particolare chinase.

Terza forma normale

È abbastanza immediato rendersi conto che tutte le relazioni del database rispettano la terza forma normale, infatti tutti gli attributi non-primi presenti nelle relazioni dipendono *solo ed esclusivamente* dalla chiave primaria, escludendo, quindi,

ogni possibile dipendenza transitiva. In altre parole, nessun attributo dipende da altri attributi non-chiave.

Forma normale di Boyce e Codd

Nessuna delle relazioni presenti nel nuovo database possiede più di una chiave candidata, quindi non possono esserci due o più chiavi sovrapposte. Ciò è sufficiente a garantire che, se le relazioni sono in 3NF, allora rispettano anche la BCNF.

Poichè tutte le relazioni rispettano queste condizioni si può affermare che il database è in forma normale di Boyce e Codd (*BCNF*).

3.1.4 Simmetria ed estendibilità del nuovo database

Il nuovo database presenta forti caratteristiche di simmetria delle entità presenti. Questo consente una facile estendibilità nel caso si decidesse di aggiungere nuove sorgenti dati, ovvero integrare un nuovo database con i quattro già utilizzati.

In particolare per tutti i database esistono le entità di tipo *proteina* e *sito* collegate tra loro. Queste due entità costituiscono il nucleo del database, perciò ogni altra entità presente fa riferimento o è collegata ad una di queste due. Ogni nuova entità proteina di un nuovo database va collegata con l'entità *Protein* e lo stesso vale per i siti, che vanno collegati a *Site*; in questo modo si tiene traccia delle sorgenti delle informazioni presenti nelle proteine e nei siti unificati. Inoltre se nelle nuove sorgenti dati sono presenti informazioni sulle specie e/o sulle strutture dei PDB è sufficiente collegare la nuova entità delle proteine con le entità *Species* e *PDB* già presenti. Un discorso analogo vale per i dati PubMed e anche per le informazioni sulle chinasi e sui gruppi di chinasi.

In pratica, quindi, per aggiungere una nuova sorgente dati, basta *estendere* il database con nuove relazioni, senza dover modificare quelle già presenti, se non, eventualmente, per qualche piccolo dettaglio. In particolare, per ogni nuovo database vanno introdotte, oltre alle relazioni "di base", come le proteine o i siti, alcune relazioni di collegamento come quelle che rappresentano i legami tra un sito di fosforilazione e le chinasi che lo fosforilano o quelle che servono a collegare i siti con gli ID PubMed.

Non è possibile stabilire a priori con esattezza quante e quali relazioni sarà necessario introdurre perchè questo dipende dal tipo di dati memorizzati nella nuova sorgente. Per esempio potrebbero esserci dati sui gruppi di chinasi, ma non sulle singole chinasi, o potrebbero mancare le informazioni sulle strutture dei PDB. Le relazioni che di certo non possono mancare sono i collegamenti con *Protein* e con *Site*, tutto il resto è opzionale.

3.2 Alcuni esempi di query eseguibili sul nuovo database

In questa sezione vengono presentati alcuni esempi di query con cui è possibile interrogare il nuovo database. Attraverso questi esempi si può verificare la potenza espressiva del database e la sua effettiva utilità. Nel caso di query parametriche viene usato un valore di esempio.

3.2.1 Query #1

Estrarre tutti i siti di fosforilazione di una determinata proteina e mostrarne le informazioni sulla relativa specie.

```
1 SELECT P.ourID, CONCAT(SUBSTR(P.PrimarySequence, 1, 20), '...') AS
      PrimarySequence, P.PriSeq_Source, S.Position, S.PhosphorylationType
      AS Phospho, S.Site_source, SP.TaxID, SP.LatinName, SP.CommonName
2
3 FROM Protein AS P INNER JOIN Site AS S ON P.ourID = S.ourID INNER JOIN
      Species AS SP ON P.Species = SP.TaxID
4
5 WHERE P.ourID = 116;
```

Codice 3.1: Query #1

ourID	PrimarySequence	PriSeq_Source	Position	Phospho	Site_source	TaxID	LatinName	CommonName
116	MSFTPGKQSSSRASSGNRSG...	UniProt	167	S	UniProt	233261	Bovine coronavirus	strain LSU-94LSS-051
116	MSFTPGKQSSSRASSGNRSG...	UniProt	174	T	UniProt	233261	Bovine coronavirus	strain LSU-94LSS-051
116	MSFTPGKQSSSRASSGNRSG...	UniProt	390	S	UniProt	233261	Bovine coronavirus	strain LSU-94LSS-051
116	MSFTPGKQSSSRASSGNRSG...	UniProt	423	S	UniProt	233261	Bovine coronavirus	strain LSU-94LSS-051
116	MSFTPGKQSSSRASSGNRSG...	UniProt	427	T	UniProt	233261	Bovine coronavirus	strain LSU-94LSS-051

5 rows in set (0.00 sec)

Figura 3.3: Risultato della query #1

3.2.2 Query #2

Trovare tutte le chinasi che fosforilano una determinata proteina.

```sql
SELECT DISTINCT K.*
FROM Kinase_Site AS KS INNER JOIN Kinase AS K ON KS.KinaseID = K.OurID
WHERE KS.OurID = 374180;
```

Codice 3.2: Query #2

OurID	KinaseName
381110	PRKCB
29305	TEC
368703	ITK
374180	BTK
490892	ABL1
20191	LYN
116556	SYK

7 rows in set (0.00 sec)

Figura 3.4: Risultato della query #2

3.2.3 Query #3

Trovare tutti i siti di fosforilazione per una determinata proteina per i quali esistono informazioni sulle chinasi.

```sql
SELECT *
FROM Kinase_Site
WHERE OurID = 374180;
```

Codice 3.3: Query #3

OurID	Position	KinaseID	RelationType
374180	180	381110	
374180	223	29305	
374180	223	368703	
374180	223	374180	SIM
374180	223	490892	
374180	551	20191	
374180	551	116556	
374180	551	374180	SIM

8 rows in set (0.00 sec)

Figura 3.5: Risultato della query #3

3.2.4 Query #4

Trovare tutti i siti di fosforilazione che sono fosforilati da almeno 8 chinasi, mostrare le informazioni sulla proteina, sul sito, sulle chinasi e sulla specie, ordinate prima per ID della proteina, poi per posizione del sito e infine per nome della chinase.

```
1  SELECT P.OurID AS ProteinID, CONCAT(SUBSTR(P.PrimarySequence, 1, 20),
       '...') AS PrimarySequence, S.Position AS Pos, S.PhosphorylationType
       AS Phospho, K.KinaseName AS KinaseName, KS.RelationType AS Type,
       SP.LatinName AS Species
2
3  FROM Kinase AS K INNER JOIN Kinase_Site AS KS ON K.OurID = KS.KinaseID
       INNER JOIN Site AS S ON KS.OurID = S.OurID AND KS.Position =
       S.Position INNER JOIN Protein AS P ON S.OurID = P.OurID INNER JOIN
       Species AS SP ON P.Species = SP.TaxID
4
5  WHERE (P.OurID, S.Position) IN (
6     SELECT KS.OurID, KS.Position
7     FROM Kinase_Site AS KS
8     GROUP BY KS.OurID, KS.Position
9     HAVING COUNT(*) > 8
10 )
11 ORDER BY ProteinID ASC, Pos ASC, KinaseName ASC;
```

Codice 3.4: Query #4

ProteinID	PrimarySequence	Pos	Phospho	KinaseName	Type	Species
30044	MAEPRQEFEVMEDHAGTYGL...	262	S	CAMK2A		Homo sapiens
30044	MAEPRQEFEVMEDHAGTYGL...	262	S	GSK3B		Homo sapiens
30044	MAEPRQEFEVMEDHAGTYGL...	262	S	MAPK12		Homo sapiens
30044	MAEPRQEFEVMEDHAGTYGL...	262	S	MARK1		Homo sapiens
30044	MAEPRQEFEVMEDHAGTYGL...	262	S	PHKG1		Homo sapiens
30044	MAEPRQEFEVMEDHAGTYGL...	262	S	PKN1		Homo sapiens
30044	MAEPRQEFEVMEDHAGTYGL...	262	S	PRKACA		Homo sapiens
30044	MAEPRQEFEVMEDHAGTYGL...	262	S	PRKCD		Homo sapiens
30044	MAEPRQEFEVMEDHAGTYGL...	262	S	RPS6KB1		Homo sapiens
111863	MSQWYELQQLDSKFLEQVHQ...	701	Y	EGFR		Homo sapiens
111863	MSQWYELQQLDSKFLEQVHQ...	701	Y	FGFR3		Homo sapiens
111863	MSQWYELQQLDSKFLEQVHQ...	701	Y	FGFR4		Homo sapiens
111863	MSQWYELQQLDSKFLEQVHQ...	701	Y	JAK1		Homo sapiens
111863	MSQWYELQQLDSKFLEQVHQ...	701	Y	JAK2		Homo sapiens
111863	MSQWYELQQLDSKFLEQVHQ...	701	Y	KIT		Homo sapiens
111863	MSQWYELQQLDSKFLEQVHQ...	701	Y	LCK		Homo sapiens
111863	MSQWYELQQLDSKFLEQVHQ...	701	Y	PTPN11		Homo sapiens
111863	MSQWYELQQLDSKFLEQVHQ...	701	Y	PTPN2		Homo sapiens
111863	MSQWYELQQLDSKFLEQVHQ...	701	Y	TYK2		Homo sapiens
279663	MSGGGPSGGGPGGSGRARTS...	21	S	AKT1		Homo sapiens
279663	MSGGGPSGGGPGGSGRARTS...	21	S	PRKACA		Homo sapiens
279663	MSGGGPSGGGPGGSGRARTS...	21	S	PRKCA		Homo sapiens
279663	MSGGGPSGGGPGGSGRARTS...	21	S	PRKCB		Homo sapiens
279663	MSGGGPSGGGPGGSGRARTS...	21	S	PRKCD		Homo sapiens
279663	MSGGGPSGGGPGGSGRARTS...	21	S	PRKCG		Homo sapiens
279663	MSGGGPSGGGPGGSGRARTS...	21	S	PRKD3		Homo sapiens
279663	MSGGGPSGGGPGGSGRARTS...	21	S	SGK1		Homo sapiens
279663	MSGGGPSGGGPGGSGRARTS...	21	S	SGK3		Homo sapiens
437863	MTMESGAENQQSGDAAVTEA...	133	S	AKT1		Homo sapiens
437863	MTMESGAENQQSGDAAVTEA...	133	S	MAPKAPK2		Homo sapiens
437863	MTMESGAENQQSGDAAVTEA...	133	S	PRKG1		Homo sapiens
437863	MTMESGAENQQSGDAAVTEA...	133	S	RPS6KA1		Homo sapiens
437863	MTMESGAENQQSGDAAVTEA...	133	S	RPS6KA2		Homo sapiens
437863	MTMESGAENQQSGDAAVTEA...	133	S	RPS6KA3		Homo sapiens
437863	MTMESGAENQQSGDAAVTEA...	133	S	RPS6KA4		Homo sapiens
437863	MTMESGAENQQSGDAAVTEA...	133	S	RPS6KA5		Homo sapiens
437863	MTMESGAENQQSGDAAVTEA...	133	S	SGK1		Homo sapiens
437863	MTMESGAENQQSGDAAVTEA...	133	S	TSSK4		Homo sapiens

38 rows in set (19.26 sec)

Figura 3.6: Risultato della query #4

3.2.5 Query #5

Trovare tutti i siti di fosforilazione di tutte le proteine fosforilate da una certa chinase, includere le informazioni sulla specie di appartenenza della chinase.

```sql
1 SELECT S.OurID, P.Species AS P_SpeciesID, S.Position,
     S.PhosphorylationType AS Phospho, S.Site_Source, KS.KinaseID,
     KS.RelationType AS Type, SP.TaxID AS K_SpeciesID, SP.LatinName,
     Sp.CommonName
2
3 FROM Kinase_Site AS KS INNER JOIN Site AS S ON KS.OurID = S.OurID AND
     KS.Position = S.Position INNER JOIN Protein AS P ON P.OurID =
     KS.KinaseID INNER JOIN Species AS SP ON SP.TaxID = P.Species
4
5 WHERE KS.KinaseID = 20034;
```

Codice 3.5: Query #5

OurID	P_SpeciesID	Position	Phospho	Site_Source	KinaseID	Type	K_SpeciesID	LatinName	CommonName
10604	9606	338	Y	HPRD = ELM	20034		9606	Homo sapiens	Human
10604	9606	419	Y	UniProt = PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
10604	9606	530	Y	UniProt = PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
20191	9606	397	Y	UniProt = PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
37742	9606	628	Y	PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
91638	9606	643	Y	PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
92672	9606	537	Y	UniProt = PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
142251	9606	1197	Y	UniProt = PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
167662	9606	394	Y	UniProt = PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
350919	9606	340	Y	UniProt = PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
362508	9606	570	Y	UniProt = PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
411753	9606	663	Y	HPRD	20034		9606	Homo sapiens	Human
411753	9606	690	Y	UniProt = PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
411753	9606	713	Y	UniProt = PTM = HPRD = ELM	20034		9606	Homo sapiens	Human
426584	9606	27	Y	HPRD	20034		9606	Homo sapiens	Human
426584	9606	44	Y	HPRD	20034		9606	Homo sapiens	Human
426584	9606	360	Y	HPRD	20034		9606	Homo sapiens	Human
449683	9606	453	Y	HPRD	20034		9606	Homo sapiens	Human
485689	9606	762	Y	UniProt = PTM = HPRD	20034		9606	Homo sapiens	Human
485689	9606	822	Y	UniProt = PTM = HPRD	20034		9606	Homo sapiens	Human

20 rows in set (0.00 sec)

Figura 3.7: Risultato della query #5

3.2.6 Query #6

Trovare tutte le informazioni sulle strutture PDB di una certa proteina.

```
SELECT P.OurID, PDB.*

FROM Protein AS P INNER JOIN Protein_PDB AS PPDB ON P.OurID = PPDB.OurID
    INNER JOIN PDB ON PPDB.PDBID = PDB.PDBID

WHERE P.OurID = 374180;
```

Codice 3.6: Query #6

OurID	PDBID	ExperimentType	ExperimentResolution	Chains
374180	1AWW	NMR	-	A=212-275
374180	1AWX	NMR	-	A=212-275
374180	1B55	X-ray	2.40 A	A/B=2-170
374180	1BTK	X-ray	1.60 A	A/B=2-170
374180	1BWN	X-ray	2.10 A	A/B=2-170
374180	1K2P	X-ray	2.10 A	A/B=397-659
374180	1QLY	NMR	-	A=216-273
374180	2GE9	NMR	-	A=270-386
374180	2Z0P	X-ray	2.58 A	A/B/C/D=2-170
374180	3GEN	X-ray	1.60 A	A=382-659
374180	3K54	X-ray	1.94 A	A=382-659
374180	3OCS	X-ray	1.80 A	A=393-656
374180	3OCT	X-ray	1.95 A	A=393-656
374180	3P08	X-ray	2.30 A	A/B=393-659

14 rows in set (0.05 sec)

Figura 3.8: Risultato della query #6

Capitolo 4

Struttura del codice del programma

4.1 Modularità del programma

L'intero progetto è stato sviluppato in modo da essere modulare: se due concetti sono logicamente separati, come lo sono per esempio parsing dei file originali e popolamento del database, essi vanno implementati mantenendo questa divisione logica, ad esempio codificando parser e populator in due classi distinte.
Questo offre alcuni vantaggi:

- migliora la leggibilità generale del progetto;
- rende molto più facile l'individuazione e la correzione di errori eseguendo test isolati dei singoli componenti;
- ciascun modulo può essere sviluppato indipendentemente dagli altri, una volta che siano stabilite le interazioni tra di essi.

4.1.1 Separazione tra classi, interfacce ed eccezioni

La modularità del codice non è relativa ai soli tre moduli principali concettualmente divisi: ogni componente del progetto dispone di una propria interfaccia che ne definisce le interazioni con gli altri componenti e/o con il mondo esterno a livello astratto. Esiste poi una classe che implementa una specifica interfaccia dove vengono definiti tutti i dettagli realizzativi delle funzioni dichiarate dall'interfaccia stessa.
Questo permette di scrivere un modulo che ne usa un altro anche se quest'ultimo non è ancora stato effettivamente scritto, basta che ne sia nota l'interfaccia. In questo modo, inoltre si può modificare un'implementazione per migliorarne le prestazioni e/o correggerne gli errori senza dover modificare anche i metodi che usano

gli oggetti che sono stati modificati.
Questa divisione classi/interfacce funziona molto bene adottando una strategia si sviluppo di tipo Top-Down, dove si ha già una chiara visione d'insieme del problema.

Sono state aggiunte anche alcune eccezioni che si limitano ad estendere l'eccezione base di Java con un messaggio di errore specifico per rendere più chiaro il problema.

Come già accennato l'intero progetto è stato svolto creando un programma Java diviso in tre moduli concettualmente separati, ma interagenti tra di loro.

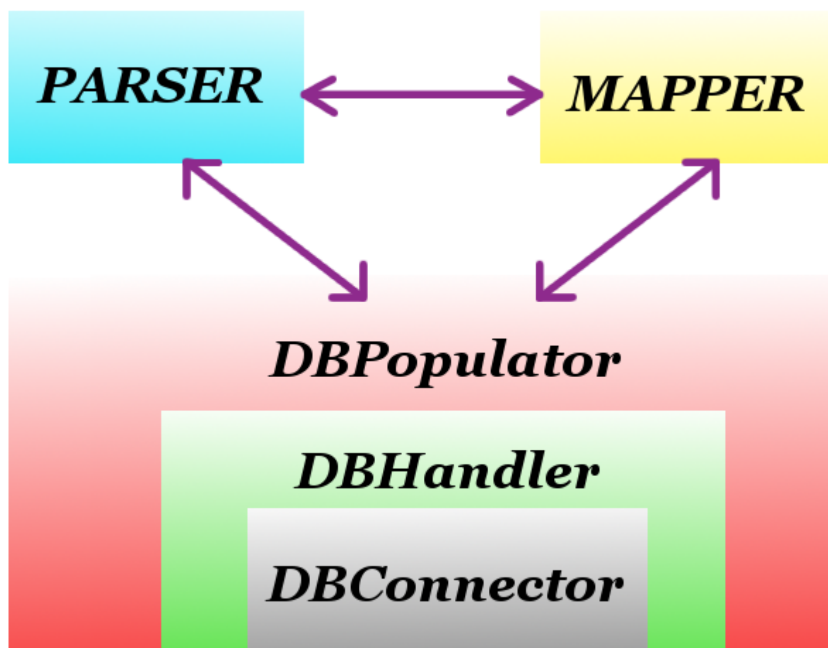

Figura 4.1: Schema dei moduli del programma.

1. Il primo modulo è il ***Parser***, il cui scopo è quello di elaborare i file originali dei quattro database scelti per produrre dei file che siano utilizzabili dagli altri due moduli del programma;

2. il secondo componente è il ***Mapper***, la cui funzione è quella di analizzare gli ID delle proteine dei quattro database e cercare i collegamenti presenti tra gli stessi. Questi ID vengono poi collegati a un nuovo ID, creato apposta per il nuovo database, attraverso il quale è possibile uniformare tutte le informazioni;

3. l'ultimo modulo del programma è il ***Populator***: è il componente che si occupa di inserire i dati elaborati dagli altri due moduli nel nuovo database. A sua volta, il modulo Populator è concettualmente diviso in tre strati:

(a) lo strato più basso è il *Connector* e si occupa di gestire la connessione con il DB o il DBMS (nel caso non esista ancora un DB a cui connettersi);

(b) lo strato intermedio è l'*Handler*, il quale fornisce metodi per il popolamento di singole tabelle, l'aggiunta/eliminazione di tabelle, l'eliminazione di tutte le righe da una tabella e l'inserimento multiplo di valori, indispensabile per ottenere prestazioni accettabili quando si inseriscono così tanti dati;

(c) infine lo strato superiore, chiamato anch'esso *Populator* si occupa di popolare le varie tabelle del database leggendo i dati dai file creati dai moduli Parser e Mapper.

4.2 Combinazione degli approcci Bottom-Up e Top-Down

Nell'affrontare questo progetto si è scelto di seguire, inizialmente, una metodologia di tipo *Bottom-Up*: ovvero partire da un dettaglio dell'intero problema, analizzarlo a fondo e scrivere codice funzionante per quel particolare sotto-problema. Il passo successivo consiste nell'unire tutti i sotto-programmi così ottenuti per raggiungere lo scopo finale.
Questa strategia era molto adatta alla situazione perché inizialmente non era chiaro tutto il contesto e tutte le specifiche, così ci si è concentrati su alcune parti relative al parsing dei file analizzandole in dettaglio.
Ciò ha permesso di scrivere codice specifico per un singolo sotto-problema (ad esempio il parsing del file originale del database PTM) testabile fin da subito, ma con l'incognita di non sapere esattamente come poi questo codice dovesse essere integrato con il resto del programma, con il rischio, cioè, di dover modificare più volte quanto già scritto.

Si è continuato a seguire questa strada fino a quando il progetto era quasi completo, a quel punto, avendo acquisito una visione completa ed esaustiva del problema in tutte le sue sfaccettature è stato possibile svolgere un'operazione di *refactoring* del codice: usando questa volta una metodologia di tipo *Top-Down* è stato riprogettato l'intero programma rendendolo ancora più modulare e leggibile. Ad esempio nella prima versione esistevano solo due moduli, Parser e Populator, in quanto inizialmente non si era fatta distinzione tra parsing dei files e mapping degli ID.

Il vantaggio principale di questo secondo approccio è che, sapendo chiaramente cosa dovesse essere fatto, si è potuto progettare l'intero programma avendo come priorità le interazioni tra i vari moduli, senza pensare a come sono implementati, ovvero un approccio a scatole chiuse.
È stato quindi molto facile riutilizzare il codice scritto precedentemente, con alcune opportune modifiche, con questa nuova struttura per ottenere il risultato finale desiderato avendo la certezza che le singole parti fossero funzionanti grazie ai test precedenti e dovendo testare solo le modifiche e l'integrazione.

4.3 Analisi del modulo Parser

Questo modulo è composto da sei interfacce, più altrettante classi che le implementano. Non sono previste eccezioni specifiche per questo modulo. Vengono qui presentate le sole interfacce con alcuni accenni alle relative implementazioni, per maggiori dettagli sul codice si rimanda al Javadoc. Cinque delle sei classi/interfacce sono di supporto, mentre l'ultima svolge le funzioni di parsing vere e proprie.

4.3.1 L'interfaccia Species

Questa interfaccia serve a modellare l'oggetto "specie", il quale è composto fondamentalmente da tre parametri:

1. un ID tassonomico, con il quale identificare la specie. È lo stesso che viene usato dal database **NCBI**, ma per alcune specie tale ID non era presente nei file originali quindi è stato temporaneamente sostituito da un valore negativo (per maggiori dettagli si veda la sezione 6.4.1 del capitolo sugli sviluppi futuri);

2. il nome Latino della specie, usato in letteratura scientifica;

3. il nome comune, se presente, oppure le informazioni testuali aggiuntive, a seconda dei casi.

Nota: per le specie sprovviste di ID tassonomico viene memorizzata solo un'informazione di tipo nome Latino anche se in realtà si tratta di un nome comune.

Metodi dell'interfaccia Species

Questa interfaccia è relativamente semplice e i suoi metodi sono solo i **getter** e **setter** per i parametri che caratterizzano una determinata specie: ID, nome latino e nome comune.
Per la lista completa di tali metodi, e per la relativa descrizione si rimanda alla documentazione Javadoc (si veda anche il capitolo 5 sulla documentazione).

4.3.2 L'interfaccia SpeciesHandler

Questa interfaccia serve a gestire una collezione di oggetti "specie", nell'implementazione corrente dell'interfaccia tale collezione è realizzata usando una HashMap che mappa un ID tassonomico in un oggetto "specie". Questa interfaccia risulta fondamentale perché le informazioni sulle specie non sono contenute in un solo database, quindi con un oggetto di tipo "SpeciesHandler" è possibile gestire tutte le specie incontrate man mano che si prosegue nel parsing dei quattro file.

Metodi dell'interfaccia SpeciesHandler

- `int addSpecies (Species)`: serve ad aggiungere una nuova specie alla collezione, se la specie è già presente il metodo non avrà alcun effetto pratico. Una specie è considerata già inclusa nella collezione se la collezione di Species contiene già una entry con lo stesso ID tassonomico della nuova specie. Il metodo ritorna l'ID tassonomico della specie appena inserita, oppure -1 se la specie era già presente;
- `Species getSpecies(int)`: questo metodo getter restituisce la specie cui corrisponde l'ID tassonomico passato come parametro. Se non viene trovata una corrispondenza il metodo restituisce `null`;
- `Species findLatin(String)`: questo metodo funziona come il getSpecies, ma esegue la ricerca basandosi sul nome latino della specie anziché sull'ID;
- `int getSize()`: restituisce il numero di specie presenti nella collezione;
- `int writeToFile (File, String)`: questo metodo si occupa di scrivere sul file passato come parametro una rappresentazione della collezione di specie. Il valore di ritorno è il numero di specie effettivamente scritte su file. Il parametro `String` è il separatore che verrà usato nel file per separare i tre campi della specie (ID e nomi);
- `int readFromFile (File, String)`: questo metodo svolge una funzione opposta al metodo precedente, infatti si occupa di creare una collezione di specie leggendo i dati dal file specificato in ingresso, che usa come separatore la String passata come parametro. Il valore di ritorno indica il numero di specie lette dal file;
- `int getFakeID ()`: questo metodo restituisce il prossimo fake ID da usare per inserire una nuova specie. Il fake ID è un ID tassonomico negativo che cresce verso il basso e che viene assegnato a quelle specie per le quali non era presente un ID nei file dei database.

4.3.3 L'interfaccia PDB

Questa interfaccia viene usata per modellare l'oggetto "PDB", si tratta della struttura terziaria di una proteina, ovvero la sua "forma tridimensionale". I dati che un oggetto "PDB" deve memorizzare sono:

- l'ID della proteina, in formato PDB, utilizzabile nel database *Protein Data Bank*;
- il tipo di esperimento usato per ottenere i dati e la relativa risoluzione dello stesso;
- le catene di aminoacidi coinvolte.

Metodi dell'interfaccia PDB

L'interfaccia prevede metodi getter e setter per l'ID PDB, il tipo di esperimento, la risoluzione dell'esperimento e per le catene. Queste ultime sono rappresentate internamente con un `ArrayList` di coppie (catene, aminoacidi). Queste coppie sono rappresentate usando una classe utilty `Pair` appositamente creata che verrà illustrata in seguito. Oltre a tali metodi esiste anche il metodo `void addChain (Pair<String, String>)` che aggiunge una singola catena all'oggetto PDB. Non viene eseguito nessun controllo sulla presenza precedente di una catena uguale, quindi questo metodo può provocare duplicazioni.

4.3.4 L'interfaccia PDBHandler

Questa interfaccia serve a gestire una collezione di oggetti "PDB", svolgendo un ruolo analogo a quello dell'interfaccia SpeciesHandler, anche se è molto più semplice di quest'ultima. In questo caso infatti, non c'è la necessità di leggere un file PDB per ricreare la collezione di oggetti in quanto l'unico database, tra i quattro utilizzati, contenente informazioni PDB è l'UniProt. Nell'implementazione attuale anche questa collezione è gestita attraverso una HashMap che mappa l'ID PDB nell'oggetto PDB corrispondente.

Metodi dell'interfaccia PDBHandler

- `String addPDB (PDB)`: questo metodo serve ad aggiungere un oggetto PDB alla collezione, il valore di ritorno è l'ID dell'oggetto appena inserito. Se era già presente un oggetto PDB con lo stesso ID, le informazioni precedenti vengono sovrascritte da quelle nuove;

- `void writeToFile (File)`: questo metodo server per scrivere su un file, passato come parametro, i dati PDB contenuti nella collezione. Il metodo non ha bisogno di un parametro che indichi il separatore da usare perché si suppone che l'oggetto PDB fornisca un adeguato metodo `toString()`;

- `int getSize()`: reistituisce il numero di oggetti PDB presenti nella collezione.

4.3.5 L'interfaccia UniProtRecord

A differenza dei file degli altri tre database utilizzati, il database UniProt contiene record memorizzati su molteplici righe e con un formato complesso (per maggiori dettagli si veda la sezione 2.1.1 del capitolo sui database). Si è quindi resa necessaria l'introduzione di un'interfaccia che potesse modellare un singolo record UniProt. Lo scopo di questa interfaccia è quindi quello di raccogliere le varie informazioni contenute nel file UniProt per uno specifico record a mano a mano che vengono lette e di convertirle in un record mono-riga uniforme a quelli degli altri file prodotti dal parsing.

Metodi dell'interfaccia UniProtRecord

Molti dei metodi dell'interfaccia UniProtRecord di fatto non servono, e nell'implementazione attuale non eseguono alcuna operazione, ma sono stati inclusi perché in futuro potrebbe essere necessario estrarre informazioni che per ora vengono tralasciate. Vengono qui riportati solamente i metodi che, allo stato attuale, sono effettivamente implementati e utilizzati. Per un elenco completo, comprese le descrizioni, si rimanda alla documentazione Javadoc.

- `String getValues ()`: questo metodo restituisce una stringa contenente tutte le informazioni memorizzate nel record UniProt, dovrebbe essere richiamato solo dopo aver letto e aggiunto tutte le informazioni di un dato record;
- `String setIDInfo (String)`: questo metodo usa come input la stringa `ID` del file originale UniProt, ne elabora le informazioni estraendo ID, stato e lunghezza della proteina. Il valore di ritorno è una stringa con le informazioni estratte nello stesso formato con cui verrano scritte nel file. Ovvero nello stesso formato restituito da `getValues()`;
- `String addAcessionNumbers(String)`: questo metodo usa come input la stringa `AC` del file originale UniProt e ne elabora le informazioni estraendo gli *Accession Numbers* trovati. Il valore di ritorno è una stringa con le informazioni estratte nello stesso formato con cui verrano scritte nel file. Ovvero nello stesso formato restituito da `getValues()`. L'informazione sugli AC di un record può essere memorizzata su più righe;
- `int addCalssificationReference (String)`: questo metodo viene usato per aggiungere al record le informazioni contenute nelle stringhe `OX` del file originale, tale informazione indica la specie di appartenenza della proteina e perciò ne esiste al più una per ogni recod. Il valore di ritorno è l'ID trovato.
- `String addLiteratureCitation (String, String, String, String, String, String, String, String)`: questo metodo si occupa di memorizzare le informazioni relative a citazioni di letteratura presenti nel file, si tratta delle righe con codice `R*` dove `*` può essere una lettera tra (`N`, `P`, `C`, `X`, `G`, `A`, `T`, `L`) a seconda di cosa viene memorizzato. Allo stato attuale le uniche informazioni che ci interessano sono gli ID PubMed presenti nelle linee RX ed infatti nell'implementazione attuale solo il parametro relativo alla stringa `RX` viene utilizzato. Anche in questo caso le informazioni possono essere contenute su più righe nel file originale, questo metodo può pertanto essere usato in due modi:
  - richiamato ogni volta che si incontra una riga `RX` nel file originale;
  - richiamato una sola volta passando come parametro una stringa contenente tutte le stringhe `RX` trovate concatenate.

- `String addFeatureTable (String)`: questo metodo viene usato per aggiungere le informazioni contenute nelle righe `FT` del file originale. Queste righe contengono le informazioni riguardanti i siti di fosforilazione. Il valore di ritorno è la stringa generata, la stessa restituita dal metodo `getValues()`, questa stringa contiene anche le eventuali informazioni relative alle chinasi e viene quindi salvata in un file di supporto per un uso successivo. Le informazioni possono essere contenute su più righe ed è necessario che questo metodo venga invocato passando come parametro una stringa contenente tutti i dati `FT` concatenati;

- `String addPrimarySequence(String)`: questo metodo serve per salvare nel record la sequenza primaria di aminoacidi della proteina. Nel file originale la sequenza è salvata su più righe, questo metodo richiede che queste righe vengano concatenate e passate come singola stringa. Il valore di ritorno è una stringa contenente la sequenza primaria senza spazi e su una sola riga;

Come già detto per tutti gli altri metodi si rimanda alla documentazione Javadoc.

4.3.6 L'interfaccia Parser

Questa interfaccia fornisce i metodi necessari ad analizzare i file originali dei quattro database utilizzati. Ci sono quattro metodi principali, uno per ogni database, che producono un file elaborato per ciascun database, più altri metodi che si occupano di specifici aspetti di un database o di informazioni generiche, ad esempio gli ID PubMed.

Metodi dell'interfaccia Parser

- `void parseFiles(File)`: questo è un metodo wrapper il cui unico scopo è quello di richiamare tutti gli altri metodi necessari ad eseguire il parsing nella sua interezza invocando i metodi nel giusto ordine e con i giusti parametri. Tali parametri vengono acquisiti dal file di configurazione principale passato come parametro. Per una descrizione del file di configurazione si rimanda all'apposita sezione;

- `void parseUniProt (File, File, File, File, File, File, File, File)`: questo metodo si occupa di eseguire il parsing del file originale UniProt creando, oltre al file principale elaborato, una serie di altri file di supporto. Tali file contengono i dati grezzi sulle chinasi, i dati grezzi sulle specie e i dati grezzi sui PDB; tutti questi file andranno elaborati da altri metodi. Vengono poi creati altri file di supporto, non tutti indispensabili, ma tutti utili per una migliore leggibilità del progetto, si tratta di file di mapping tra ID UniProt e ID PubMed / sequenze primarie / AC UniProt;

- `void parseUniProtKinase (File, File, File, File)`: questo metodo prende in input il file con i dati grezzi sulle chinasi e genera tre nuovi file elaborando le informazioni del file di input. I file creati contengono:
  - le informazioni certe sulle chinasi, ovvero quelle proteine che si autofosforilano;
  - le informazioni incerte, per le quali cioè non è possibile dire con certezza se si tratti di chinasi o gruppi di chinasi;
  - le informazioni sconosciute, quelle scritte in un formato misto, che richiederanno sviluppi ulteriori del programma (si veda la sezione 6.2 del capitolo sugli sviluppi futuri).
- `void parseUniProtPDB (File, File, File)`: questo metodo prende in input il file con i dati grezzi sui PDB e genera due nuovi file, uno che associa ID UniProt con ID PDB e l'altro che contiene tutte le informazioni sui PDB in un formato leggibile al Populator;
- `void parseUniProtSpecies (File, File)`: questo metodo elabora il file con i dati grezzi sulle specie creando un file in un formato utilizzabile dai moduli del programma;
- `void parsePTM (File, File, String, File, File, File)`: questo metodo si occupa di eseguire il parsing del file originale PTM creando, oltre al file principale elaborato, alcuni altri file di supporto. In particolare, viene creato un file contenente i dati grezzi sulle chinasi ed un file di supporto, che memorizza il mapping tra ID PTM e ID PubMed;
- `void parsePTMKinase (File, File, File, File, File)`: questo metodo svolge le stesse funzioni del metodo `parseUniProtKinase`, ma con una differenza. Nel database PTM è possibile identificare con certezza alcuni gruppi di chinasi, quindi questo metodo genera un file anche per i gruppi;
- `void parseELM (File, File, String, File, File, File, File, File)`: questo metodo si occupa di eseguire il parsing del file originale ELM creando, oltre al file principale elaborato, una serie di altri file di supporto. Tali file contengono i dati grezzi sulle chinasi e i dati grezzi sulle specie che andranno elaborati da altri metodi. Vengono poi creati altri file di supporto, non tutti indispensabili, ma tutti utili per una migliore leggibilità del progetto, si tratta di file di mapping tra sito ELM e ID PubMed, AC ELM e sequenza primaria e AC ELM e ID ELM;
- `void parseELMKinase (File, File, File)`: questo metodo svolge funzioni analoghe ai precedenti `parseUniProtKinase` e `parsePTMKinase`, ma dal database ELM si possono ricavare informazioni certe solo gruppi, tutto il resto viene inserito nel file con le informazioni incerte;

- `void parseELMSpecies (File, File, File)`: questo metodo elabora il file con i dati grezzi delle specie ELM, ma richiede in input anche il file delle specie prodotto da `parseUniProtSpecies`. Infatti nel database ELM sono presenti solo i nomi latini per le specie, quindi questo metodo aggiunge al file delle specie solo quelle che non sono già presenti, usando un ID tassonomico falso (cioè negativo) e lasciando il nome comune vuoto;

- `void replaceELMSpecies (File, File)`: questo metodo modifica il file elaborato da `parseELM` per sostituire i nomi latini delle specie con gli ID tassonomici presenti nel secondo file di input, ovvero il file con le informazioni sulle specie;

- `void parseHPRD (File, File, String, File, File, File)`: questo metodo si occupa di eseguire il parsing del file originale HPRD. Oltre al file principale elaborato, vengono creati altri file: uno contiene le coppie ID HPRD e Isoform, l'altro contiene il mapping tra sito HPRD e ID PubMed. Data la struttura del file HPRD viene generato anche il file contenente le informazioni sulle chinasi certe, che non necessita di nessuna ulteriore elaborazione;

- `void createPubMedFile (File, File, File, File, File)`: questo metodo usa come input i quattro file con i dati PubMed prodotti elaborando i file originali dei database e crea in output un file contenente tutti gli ID PubMed trovati nei vari file, evitando ripetizioni di ID.

4.4 Analisi del modulo Mapper

Il modulo Mapper ha il compito di collegare i vari ID dei quattro database e generare il nuovo ID usato nel nuovo database. I quattro ID vengono trattati in modo paritetico, quindi non ce n'è uno più importante degli altri, in questo modo è possibile collegare al nuovo ID uno o più ID dei quattro database, a seconda dei collegamenti trovati senza nessuna restrizione dovuta ai collegamenti reciproci. Alcuni esempi dei collegamenti generati:

Nuovo ID	ID UniProt	ID PTM	ID HPRD	AC ELM
66	TYB4_HUMAN	TYB4_HUMAN	02135	-
173396	NEP1_MOUSE	NEP1_MOUSE	-	O35130
494320	PIMT_KELP3	-	-	-
531133	-	-	01724	-
524432	CNRG_HUMAN	CNRG_HUMAN	08914	P18545
527355	-	-	-	ENSP00000373910

Il modulo Mapper è composto di due interfacce, con relativa implementazione e prevede l'uso di cinque eccezioni. Come per il Parser vengono qui presentate le sole interfacce con alcuni accenni alle relative implementazioni, per maggiori dettagli sul codice si rimanda al Javadoc.

4.4.1 L'interfaccia DBLink

Lo scopo dell'interfaccia DBLink è quello di fornire una rappresentazione di un collegamento tra gli ID dei quattro database e il nuovo ID usato nel database.

Metodi dell'interfaccia DBLink

I metodi forniti dall'interfaccia DBLink sono dei semplici getter e setter per gli ID dei quattro database, con alcune particolarità:

- per il database ELM vengono collegati gli AC, non gli ID, in quanto gli AC si riferiscono alle proteine, mentre gli ID si riferiscono ai singoli siti;
- per i database ELM e HPRD non vengono usati dei metodi setter ma vengono usati rispettivamente `addELM(String)` e `addHPRD(String)`.

Questo succede perché per gli AC ELM e gli ID HPRD possono esserci più valori che vanno collegati allo stesso nuovo ID, in questi casi i metodi eseguiranno un controllo evitando duplicazioni e inseriranno tutti i valori come stringhe concatenate separate da un apposito separatore.
I metodi getter restituiscono `null` quando l'ID richiesto è una stringa vuota, ovvero non è presente nel collegamento.

4.4.2 L'interfaccia IDMapper

Questa interfaccia fornisce i metodi necessari a mappare gli ID dei quattro database con il nuovo ID. Come per il modulo Parser esiste un metodo wrapper il cui scopo è quello di invocare tutti gli altri metodi nell'ordine e con i parametri corretti. Il mapping degli ID di ciascun database viene fatto in due fasi: prima viene letto un file che fornisce gli ID e/o gli AC per quel database, poi i dati letti vengono usati per creare il mapping in modo incrementale. Ovvero inizialmente la mappa è vuota, poi viene riempita con gli ID del primo database, successivamente viene aggiornata con gli ID degli altri database modificando o aggiungendo voci. Durante la seconda fase, per ogni database, può essere sollevata un'eccezione se la prima fase non era stata eseguita.

Metodi dell'interfaccia IDMapper

- `void mapDatabases (File)`: questo è un metodo wrapper il cui unico scopo è quello di richiamare tutti gli altri metodi necessari ad eseguire il mapping nella sua interezza invocando i metodi nel giusto ordine e con i giusti parametri. Tali parametri vengono acquisiti dal file di configurazione principale passato come parametro. Per una descrizione del file di configurazione si rimanda all'apposita sezione;

- `void readUniProtIDsACs (File)`: questo metodo riceve in input uno dei file di supporto creati durante il parsing UniProt e crea una HashMap interna per mappare gli ID UniProt negli AC corrispondenti, tale mappa verrà successivamente usata per collegare gli ID degli altri database con quelli di UniProt, passando attraverso i rispettivi AC;

- `void mapUniProt ()throws UniProtIDsNotPresentException`: questo metodo si occupa di collegare i nuovi ID con gli ID UniProt e può lanciare un'eccezione nel caso in cui gli ID UniProt non siano stati letti;

- `void readPTMIDs (File)`: questo metodo riceve in input uno dei file di supporto creati durante il parsing PTM e crea un HashSet interno contenente tutti gli ID PTM trovati, eliminando eventuali duplicati. Poichè PTM usa gli stessi ID di UniProt, il mapping tra i due database è quasi immediato;

- `void mapPTM ()throws PTMIDsSetNotPresentException`: questo metodo si occupa di collegare i nuovi ID con gli ID PTM e può lanciare un'eccezione nel caso in cui gli ID PTM non siano stati letti. Per prima cosa viene controllata la presenza dell'ID PTM come se fosse un ID UniProt e se viene trovato i due ID vengono collegati allo stesso nuovo ID, altrimenti viene aggiunta una nuova voce avente ID UniProt nullo;

- `void readELMACs (File)`: questo metodo riceve in input uno dei file di supporto creati durante il parsing ELM e crea un HashSet interno contenente tutti gli AC ELM trovati, eliminando eventuali duplicati;

- `void mapELM ()throws ELMACsSetNotPresentException`: questo metodo si occupa di collegare i nuovi ID con gli AC ELM e può lanciare un'eccezione nel caso in cui gli AC ELM non siano stati letti. Il collegamento con gli ID già presenti viene fatto eseguendo una ricerca degli AC ELM tra gli AC UniProt. Se viene trovata una corrispondenza allora la mappa viene solo aggiornata aggiungendo l'AC ELM, altrimenti viene creata una nuova entry avente ID UniProt e ID PTM entrambi nulli. Gli AC ELM non collegabili ad alcun ID UniProt vengono salvati in un HashSet per poter essere confrontati con gli AC HPRD non collegabili agli ID UniProt;

- `void readHPRDIDsACs (File, File, File)`: questo metodo è più complesso degli equivalenti per gli altri database a causa della struttura dei file HPRD. Gli input richiesti sono:

  - un file contenente tutti gli ID HPRD che sono interessati da fosforilazione, si tratta di uno dei file creati in fase di parsing;
  - il file originale "`HPRD_ID_MAPPINGS.txt`" scaricabile dal database HPRD;
  - il file con le informazioni sulle chinasi del database HPRD creato dal Parser.

  Vengono letti il primo e il terzo file ed estratti tutti gli ID HPRD, sia quelli interessati da fosforilazione, sia quelli che compaiono solo come chinasi. Infine viene usato il secondo file per creare una HashMap che colleghi gli ID HPRD con gli AC HPRD;

- `void mapHPRD ()throws HPRDIDsACsMapNotPresentException`: questo metodo si occupa di collegare i nuovi ID con gli ID HPRD e può lanciare un'eccezione nel caso in cui gli ID e gli AC HPRD non siano stati letti. Il collegamento con gli ID già presenti viene fatto eseguendo una ricerca degli AC HPRD tra gli AC UniProt. Se viene trovata una corrispondenza allora la mappa viene solo aggiornata aggiungendo l'ID HPRD, altrimenti viene creata una nuova entry avente ID UniProt, ID PTM e AC ELM tutti nulli. Gli AC HPRD non collegabili ad alcun ID UniProt vengono salvati in un HashSet per poter essere confrontati con gli AC ELM non collegabili agli ID UniProt;

- `void writeMappingFile (File)throws MainMapNotPresentException`: questo metodo viene usato per salvare su file l'HashMap contenente tutti i collegamenti tra ID creati. Il metodo può lanciare un'eccezione nel caso in cui l'HashMap sia vuota;

- `HashMap<Integer, DBLink> getMap ()`: questo è un semplice metodo getter usato per recuperare l'HashMap;

- `HashMap<Integer, DBLink> createMapFromFile(File)`: questo metodo viene usato per ricreare l'HashMap a partire dal file in cui è stata salvata. Il valore di ritorno è la mappa ricostruita.

4.5 Analisi del modulo Populator

Il modulo Populator ha il compito di creare automaticamente il nuovo database (il cui codice sorgente è salvato su file) e di popolarne tutte le tabelle con i dati elaborati dai moduli Parser e Mapper. Il modulo, come già detto, è composto da tre strati, ognuno rappresentato da un'apposita interfaccia. Esiste poi una quarta interfaccia con il compito di rappresentare i parametri di connessione al database. Il modulo Populator comprende anche un'eccezione il cui significato viene spiegato nel seguito. Come per i due moduli precedenti, vengono qui presentate le sole interfacce con alcuni accenni alle relative implementazioni, per maggiori dettagli sul codice si rimanda al Javadoc.

4.5.1 L'interfaccia DBParameters

Questa interfaccia fornisce metodi per gestire i parametri di connessione al database o al DBMS:

- nome utente e password;
- indirizzo del server;
- nome del database;
- driver da utilizzare per la connessione.

Metodi dell'interfaccia DBParameters

L'interfaccia DBParameters fornisce metodi getter e setter per controllare tutti i parametri di connessione necessari. Meritano una menzione particolare i seguenti due metodi:

- `void setAll (String, String, String, String, String)`: che imposta tutti i parametri in una sola volta. Le stringhe in ingresso sono, nell'ordine: l'URL del server, il nome del database, il driver di connessione, il nome utente e la password;
- `void setAll(File)`: questo metodo ha la stessa funzione del precedente ma legge tutti i parametri necessari dal file passato in input.

4.5.2 L'interfaccia DBConnector

L'interfaccia DBConnector fornisce i metodi necessari a stabilire una connessione con un DB o un DBMS. Si tratta dello strato più basso di gestione del database e svolge quindi una funzione basilare.

Metodi dell'interfaccia DBConnector

Nessuno dei metodi dell'interfaccia richiede esplicitamente i parametri del database, perché questi vengono memorizzati nell'implementazione come campi della classe. I metodi sono:

- `Connection connect (boolean)`: crea una connessione con il database e restituisce un oggetto che rappresenta la connessione creata. Attraverso il parametro booleano è possibile specificare se il metodo debba stampare messaggi di stato;
- `Connection connectToDBMS(boolean)`: crea una connessione con il DBMS e restituisce un oggetto che rappresenta la connessione creata. Attraverso il parametro booleano è possibile specificare se il metodo debba stampare messaggi di stato;
- `void closeConnection (boolean)`: chiude l'ultima connessione aperta. Se il parametro di ingresso è TRUE stampa dei messaggi di stato.

4.5.3 L'interfaccia DBHandler

È l'interfaccia che rappresenta lo strato intermedio di gestione del database e fornisce i metodi necessari ad aggiungere/eliminare tabelle e inserire i dati.

Metodi dell'interfaccia DBHandler

Tutti i metodi prevedono un parametro booleano, quando è impostato a TRUE il metodo stampa dei messaggi di stato. Di seguito sono riportati i metodi dell'interfaccia:

- `void createDatabase (File, boolean)`: questo metodo viene usato per creare il nuovo database con tutte le tabelle leggendo il codice sorgente da un file. Le righe vuote o quelle che iniziano con "#" vengono ignorate, ciò permette di inserire dei commenti nel file. Inoltre, come spiegato nella descrizione del file, la dimensione dei dati `VARCHAR` viene parametrizzata. Non viene effettuato nessun controllo sulla presenza precedente di un altro database con lo stesso nome di quello che si sta creando;
- `void addTable (String, boolean)`: questo metodo aggiunge una singola tabella al database eseguendo la query di creazione specificata in ingresso come stringa;
- `void truncateTable (String, boolean)`: questo metodo esegue un'operazione di `TRUNCATE` sulla tabella il cui nome è passato come parametro;
- `void deleteTableRows (String, boolean)`: questo metodo ha lo stesso effetto del metodo `truncateTable`, ma può essere richiamato anche su tabelle

riferite da altre tabelle ed è significativamente più lento dell'altro, perché in questo caso l'operazione di cancellazione deve essere annullabile, ovvero deve essere possible eseguirne il *roll-back*;

- `void deleteTable (String, boolean)`: questo metodo esegue un'operazione di `DROP TABLE` sulla tabella specificata dalla stringa;

- `int insert (String, boolean)`: questo metodo esegue l'operazione di `INSERT` specificata dalla stringa di ingresso;

- `void prepareMultiInsert (String, String[], boolean, boolean)`: questo è il metodo principale del DBHandler, il suo scopo è quello di memorizzare temporaneamente i dati da inserire per eseguire degli `INSERT` multipli. Il primo parametro è il nome della tabella da popolare, il secondo è un array di stringhe contenente i valori da inserire in ciascuna colonna, mentre il terzo viene impostato a TRUE solo quando il metodo viene invocato per l'ultima volta. Nell'implementazione attuale l'INSERT viene eseguito ogni 1000 righe.

4.5.4 L'interfaccia DBPopulator

Questa interfaccia fornisce i metodi usati per popolare le varie tabelle del database, più un metodo wrapper analogo a quelli di Parser e Mapper usato per richiamare tutti metodi di popolamento nel giusto ordine e con i giusti parametri. Il popolamento delle varie tabelle può avvenire in due modi: un alcuni casi un metodo popola solo una specifica tabella, in altri casi un solo metodo può popolare un gruppo di tabelle correlate.

Metodi dell'interfaccia DBPopulator

Tutti i metodi prevedono un parametro booleano, quando è impostato a `TRUE` il metodo stampa dei messaggi di stato. I file da cui leggere i dati da inserire non vengono passati come parametro perché si suppone siano gestiti dall'implementazione come campi della classe. Si veda la documentazione Javadoc per un approfondimento in proposito (Capitolo 5). Di seguito sono riportati i metodi dell'interfaccia:

- `void createDatabase(boolean verbosity)`: questo metodo serve per creare il nuovo database leggendo il codice da un apposito file sorgente;

- `void populateDatabase (boolean verbosity)`: questo è il metodo wrapper del modulo Populator che si occupa di popolare tutte le tabelle del database usando gli altri metodi nel giusto ordine e con i giusti parametri. L'ordine di chiamata dei metodi è particolarmente importante per il Populator per via dei vincoli di integrità referenziale presenti nel database;

- `void populatePDBTable (boolean verbosity)`: questo metodo serve per popolare la tabella base PDB con i dati generati dal Parser;

- `void populateSpeciesTable (boolean verbosity)`: questo metodo serve per popolare la tabella base Species con i dati generati dal Parser;
- `void populatePubMedTable (boolean verbosity)`: questo metodo serve per popolare la tabella base PubMed con i dati generati dal Parser;
- `void populateKinaseTable (boolean verbosity)`: questo metodo serve per popolare la tabella base Kinase, i dati utilizzati per il popolamento derivano dai file di chinasi certe dei database UniProt, PTM ed HPRD creati in fase di parsing;
- `void populateKinaseGroupTable (boolean verbosity)`: questo metodo serve per popolare la tabella base `Kinase_Group`, i dati utilizzati per il popolamento derivano dai file di gruppi di chinasi certi dei database PTM ed ELM creati in fase di parsing;
- `void populatePTMSiteKinaseGroup (boolean)`: questo metodo serve per popolare la tabella di collegamento tra siti PTM e gruppi di chinasi, può lanciare un'eccezione KinaseGroupMapNotPresentException quando la tabella `Kinase_Group` non è stata popolata, questo succede perché non esiste un file che associ i nomi dei gruppi ad un ID. Tale ID viene generato in fase di popolamento della tabella, quindi se la tabella non è stata popolata questo metodo non può riuscire a trovare gli ID dei gruppi, in quanto non esistono. Il lancio dell'eccezione "nasconde" il fallimento in inserimento che si verificherebbe a causa del vincolo di integrità referenziale;
- `void populateELMSiteKinaseGroup (boolean)`: questo metodo viene usato per popolare la tabella di collegamento tra siti ELM e gruppi di chinasi, valgono le stesse considerazioni fatte per il precedente per quanto riguarda il lancio dell'eccezione `KinaseGroupMapNotPresentException`;
- `void populateFromPTM (boolean)`: questo metodo serve per popolare diverse tabelle leggendo i dati dal file PTM principale elaborato dal Parser. Le tabelle che vengono popolate sono tre:
    1. la tabella `PTM_Protein`, contenente le informazioni sulle proteine PTM;
    2. la tabella `PTM_Site`, contenente le informazioni sui siti di fosforilazione PTM;
    3. la tabella `PTMSite_PubMed` contenente i collegamenti tra i siti PTM e gli ID PubMed.
- `void populateFromHPRD (boolean)`: questo metodo serve per popolare diverse tabelle leggendo i dati dal file HPRD principale elaborato dal Parser. Le tabelle che vengono popolate sono tre:
    1. la tabella `HPRD_Protein`, contenente le informazioni sulle proteine HPRD;

2. la tabella `HPRD_Site`, contenente le informazioni sui siti di fosforilazione HPRD;
3. la tabella `HPRDSite_PubMed` contenente i collegamenti tra i siti HPRD e gli ID PubMed.

- `void populateFromELM (boolean)`: questo metodo serve per popolare diverse tabelle leggendo i dati dal file ELM principale elaborato dal Parser. Le tabelle che vengono popolate sono tre:
  1. la tabella `ELM_Protein`, contenente le informazioni sulle proteine ELM;
  2. la tabella `ELM_Site`, contenente le informazioni sui siti di fosforilazione ELM;
  3. la tabella `ELMSite_PubMed` contenente i collegamenti tra i siti ELM e gli ID PubMed.
- `void populateFromUniProt (boolean)`: questo metodo serve per popolare diverse tabelle leggendo i dati dal file UniProt principale elaborato dal Parser. Le tabelle che vengono popolate sono solo due in quanto nel file UniProt non è specificato un collegamento tra i siti di fosforilazione e gli ID delle pubblicazioni PubMed che li riguardano:
  1. la tabella `UniProt_Protein`, contenente le informazioni sulle proteine UniProt;
  2. la tabella `UniProt_Site`, contenente le informazioni sui siti di fosforilazione UniProt.
- `void populateUniProtPubMed (boolean)`: questo metodo usa uno dei file di supporto prodotti durante la fase di parsing del file UniProt per popolare la tabella di collegamento tra proteine UniProt e ID PubMed;
- `void populateProtein (boolean)`: questo metodo usa vari file tra quelli prodotti dal Parser per popolare cinque tabelle. Le tabelle coinvolte sono: la nuova tabella Protein contenente i dati unificati sulle proteine; le quattro tabelle di collegamento tra Protein e le proteine dei singoli database. Nel popolare la nuova tabella Protein vengono analizzate le sequenze primarie presenti in UniProt ed ELM, se queste coincidono allora nella nuova proteina verrà memorizzata anche la sequenza primaria, altrimenti verrà lasciata vuota. Un campo testuale nella tabella tiene traccia di quali database sorgenti possedevano l'informazione sulla sequenza primaria;
- `void populateSite (boolean)`: questo è il metodo più complesso della classe ed usa i quattro file principali prodotti dal Parser e i file relativi a chinasi e gruppi di chinasi certi per popolare otto tabelle. Le tabelle coinvolte sono:
  1. la nuova tabella Site contenente i siti di fosforilazione unificati;

2. le quattro tabelle di collegamento tra Site e i siti dei singoli database;
3. una tabella di collegamento tra Site e le chinasi;
4. una tabella di collegamento tra Site e i gruppi di chinasi;
5. la tabella di collegamento tra Site e PubMed.

Nel popolare la nuova tabella Site vengono dapprima analizzati i quattro file principali prodotti dal Parser per estrarne tutte le informazioni relative ai siti e agli ID PubMed ad essi collegati. Il secondo passo consiste nel leggere la mappa principale tra nuovo ID ed ID dei quattro database: per ogni nuovo ID vengono letti i siti che fanno riferimento agli ID dei quattro database collegati all'ID attuale (quando presenti). Infine vengono confrontati i siti dei quattro database ed inseriti nella tabella quelli che sono uguali. Un campo testuale nella tabella tiene traccia di quali database sorgenti possedevano l'informazione su un particolare sito. A causa di shift nelle sequenze principali delle proteine tra un database e l'altro è possibile che alcuni siti vengano memorizzati in posizioni diverse anche se sono lo stesso sito. È inoltre possibile che in due posizioni uguali vengano segnalate due fosforilazioni diverse. Questo problema non viene risolto nell'implementazione attuale (si veda la sezione 6.3 sullo shift delle posizioni del capitolo sugli sviluppi futuri), tuttavia tutti i siti errati vengono scartati dall'inserimento e salvati in un file di output.
Durante l'elaborazione della tabella Site vengono anche memorizzate le righe per le tabelle di collegamento tra Site e i siti degli altri database. Usando i dati delle tabelle di collegamento viene poi creata la tabella `Site_PubMed`.
L'ultimo compito svolto dal metodo è la creazione delle tabelle di collegamento con le chinasi e i gruppi di chinasi. Anche questo metodo può lanciare l'eccezione KinaseGroupMapNotPresentException se la tabella `Kinase_Group` non è stata popolata;

- `void populateUniProtPDB (boolean)`: questo metodo viene usato per popolare la tabella di collegamento tra le proteine UniProt e i PubMed;
- `void populateHPRDSiteKinase (boolean)`: questo metodo serve a popolare la tabella di collegamento tra siti di fosforilazione HPRD e chinasi;
- `void populateUniProtSiteKinase (boolean)`: questo metodo serve a popolare la tabella di collegamento tra siti di fosforilazione UniProt e chinasi;
- `void populatePTMSiteKinase (boolean)`: questo metodo serve a popolare la tabella di collegamento tra siti di fosforilazione PTM e chinasi;

4.6 Creazione automatica del nuovo database

Come già accennato il nuovo database viene creato automaticamente leggendo il codice sorgente da un file apposito, specificato nel file di configurazione principale. Le caratteristiche rilevanti del file sorgente sono: possibilità di inserire commenti preceduti dal carattere "#"; il codice di generazione del database è scritto in SQL "quasi" standard; il "quasi" deriva dal fatto che i tipi di dato `VARCHAR` sono parametrizzati, infatti al posto della dimensione viene specificato un codice che il programma è in grado di interpretare. Questo risulta molto utile per adattare le dimensioni dei campi ai dati trovati nei file modificando una sola riga nel programma. Alternativamente è possibile specificare nel file il valore esatto e il programma non lo modificherà. Se si tentasse di creare due volte consecutive il database, infine, il programma segnalerebbe in output il fallimento della creazione, eventualmente in modo prolisso, ma senza subire un crash: l'esecuzione proseguirà come se il database fosse stato appena creato utilizzando quello già esistente. In particolare se viene aggiunta una sola tabella e ricreato il database, le vecchie tabelle verranno conservate, aggiungendo solo quella nuova. Questo comportamento non è tuttavia molto utile al di fuori della fase di sviluppo.

4.7 Tempi di esecuzione e problemi riscontrati nell'implementazione

Le versioni iniziali del programma potevano impiegare anche decine di minuti per la sola fase di parsing, questo per via della grande mole di dati da gestire e per l'uso di tecniche inadeguate. Con la versione finale del programma la situazione è migliorata drasticamente arrivando a completare le fasi di parsing e mapping nel giro di un solo minuto, ciò è stato possibile grazie ad una considerazione fondamentale: i dati da manipolare non sono ordinati, e non hanno bisogno di esserlo. Questo fatto implica la possibilità di usare delle veloci HashMap o degli HashSet per eseguire ricerche in tempo costante e rimuovere duplicati istantaneamente. La fase di mapping, che attualmente impiega una manciata di secondi, nelle prime versioni arrivava a durare quasi un'ora a causa di tutti i confronti necessari che venivano fatti scorrendo degli array contenenti centinaia di migliaia di dati (UniProt contiene più di 520000 record). Un altra strategia che si è rivelata totalmente inadeguata consisteva nel salvare in memoria tutti i dati elaborati dai file originali prima di scriverli sui file di output, ciò portava velocemente ad un esaurimento della memoria heap disponibile nella JVM. Questo problema è stato risolto quasi subito scrivendo i dati elaborati sui file di output appena pronti, liberando così lo spazio necessario per i dati successivi. Un'altra considerazione che ha migliorato le prestazioni del programma è che è più efficiente rileggere più volte lo stesso file per estrarre informazioni di tipo diverso, piuttosto che leggerlo una sola volta eseguendo operazioni molto complesse per estrarre i dati necessari. È il caso, ad esempio, del file PubMed che non viene creato a mano a mano che vengono letti i quattro file principali, ma viene creato in un secondo momento.

Un discorso del tutto diverso va fatto per la fase di popolamento: anche usando le tecniche molto veloci di elaborazione dei dati sfruttate per il parsing e il mapping, il popolamento del database è intrinsecamente lento per due motivi principali. Il primo risiede nella grande quantità di dati che vanno inseriti, il secondo, più importante, deriva dal grande numero di vincoli di integrità referenziale che il DBMS deve controllare ad ogni inserimento in determinate tabelle. Le prime versioni del modulo Populator potevano impiegare letteralmente dei giorni per popolare tutto il database. Questo problema è stato risolto introducendo una strategia di inserimento multiplo dei dati che ha permesso di contenere il tempo di popolamento entro un'ora. Questo parametro può essere migliorato drasticamente con l'uso di unità di memorizzazione di tipo SSD, arrivando ad una durata inferiore ai dieci minuti. Tuttavia l'inserimento multiplo dei dati non è esente da problemi: attualmente gli `INSERT` vengono effettuati ogni 1000 righe da inserire, questo può creare problemi se il DMBS pone un limite troppo basso alla lunghezza dei pacchetti accettabili. In particolare l'inserimento dei dati delle proteine, che comprende anche la sequenza primaria, può rivelarsi problematico perché le sequenze possono arrivare a contare anche migliaia di aminoacidi, ognuno codificato da un carattere, cioè da 1 Byte. Se ad esempio consideriamo una media di 1000 aminoacidi, per 1000 proteine ot-

teniamo un pacchetto di circa 1 Megabyte per i soli dati delle sequenze primarie, senza considerare tutto il resto (nome della tabelle, parentesi, ID, ecc). Di default MySQL ha proprio 1 Megabyte come limite per i pacchetti e, sebbene 1000 aminoacidi di media sia un numero molto alto, i problemi di taglia del pacchetto si verificano. Per questo è stato necessario modificare la variabile "`max allowed packet`" di MySQL inserendo manualmente la riga "`max_allowed_packet=16M`" nel file di configurazione.

Capitolo 5

Documentazione e considerazioni sul codice

5.1 Documentazione del codice in Javadoc

Per la stesura della documentazione del codice si è scelto di ricorrere al tool *Javadoc* di Java. La documentazione è stata prodotta a livello **private** in modo da essere il più esaustiva possibile. L'unica alternativa al Javadoc presa in considerazione è stato il Doxygen, che però è stato scartato in quanto, essendo il progetto sviluppato usando l'IDE *Eclipse*, la soluzione Javadoc era la più semplice ed efficiente.
Nella documentazione è presente una descrizione completa di tutte le interfacce, con i relativi metodi, per quanto riguarda le classi, invece, vengono illustrati alcuni dettagli implementativi dei vari metodi, nonché la funzione delle varie variabili di classe. Essendo compilato a livello private, il Javadoc, fornisce i dettagli riguardanti anche i metodi privati, non dichiarati nelle interfacce.
Oltre alla documentazione, il codice sorgente del programma è stato commentato in maniera esaustiva in ogni sua parte, quindi eventuali modifiche future risulteranno semplificate. In particolare, nei sorgenti, è spiegato in maniera puntigliosa lo scopo delle varie istruzioni utilizzate e anche la logica con cui vengono affrontati determinati problemi.

Di seguito vengono presentati alcuni esempi della documentazione prodotta e un esempio dei commenti nei file sorgenti:

Packages

Package	Description
it.unipd.phosphoDB.exceptions.mapper	
it.unipd.phosphoDB.exceptions.populator	
it.unipd.phosphoDB.interfaces.mapper	
it.unipd.phosphoDB.interfaces.parser	
it.unipd.phosphoDB.interfaces.populator	
it.unipd.phosphoDB.interfaces.util	
it.unipd.phosphoDB.phopsho	
it.unipd.phosphoDB.phopsho.mapper	
it.unipd.phosphoDB.phopsho.parser	
it.unipd.phosphoDB.phopsho.populator	
it.unipd.phosphoDB.phopsho.util	

Figura 5.1: La suddivisione in packages del programma.

popoulateFromUniProt

`public void popoulateFromUniProt(boolean verbosity)`

This method will read the parsed UniProt file line by line creating the two tables rows. Once all the table rows are ready the tables are populated. UniProt stores PubMed IDs related to the protein rather than to the site so this method doesn't have to populate an UniPrort site - PubMed relation table.

Specified by:

`popoulateFromUniProt` in interface `DBPopulator`

Parameters:

`verbosity` - If true the method will print status information.

See Also:

`DBPopulator.popoulateFromUniProt(boolean)`

Figura 5.2: Il metodo `populateFromUniProt` nella classe `PhosphoDBPopulator`.

Overview Package **Class** Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

it.unipd.phosphoDB.interfaces.populator

Interface DBConnector

All Known Implementing Classes:

PhosphoDBConnector

`public interface DBConnector`

This interface provides methods to connect to a database or a DBMS. It represents the lower layer of the database management module.

Since:

11/jul/2011

Version:

1.0

Author:

Fabio Marangon

Method Summary

Methods

Modifier and Type	Method and Description
`void`	`closeConnection(boolean verbosity)` Close the current connection to the database/DBMS.
`java.sql.Connection`	`connect(boolean verbosity)` Connect to an existing database.
`java.sql.Connection`	`connectToDBMS(boolean verbosity)` Connect to a DBMS, this method is to be used when no database has been created yet.

Figura 5.3: L'interfaccia `DBConnector`.

```java
102 // Read from file, line by line.
103 String line = "";
104 try {
105    while (line != null) {
106       line = buffer.readLine();
107       if (line != null) {
108          // Ignore comments.
109          if (line.compareTo("") != 0 && !line.startsWith("#")) {
110 
111             // Acquire configuration data.
112             String[] data = line.split("=");
113             configMap.put(data[0].toLowerCase().trim(), data[1].trim());
114          }
115       }
116    }
117 } catch (IOException e) {
118    System.out.println("*WARNING* Error: " + e.getMessage() + ".\n");
119 }
```

Codice 5.1: Esempio di codice commentato nel file sorgente *PhosphoParser.java*

5.2 Riutilizzabilità ed estensibilità del codice

Il codice relativo al parsing e al mapping è fortemente specifico del problema affrontato.
Per quanto riguarda il parsing, i vari metodi sono specifici per i singoli file dei database (che hanno formati diversi, per i dettagli si veda il Capitolo 2 sui database) quindi non è possibile riutilizzare tale codice, è tuttavia possibile partire da uno dei metodi già scritti e modificarlo per renderlo idoneo al parsing di un nuovo file di database che abbia una struttura almeno simile a quelle dei database usati finora. Questa procedura, come già evidenziato, è semplificata dai commenti presenti nei file sorgenti. Le modifiche da apportare in tali circostanze riguarderebbero solo i dettagli, non la struttura di base.
Per quanto riguarda il mapping si tratterebbe di aggiungere dei metodi specifici per il nuovo database incluso, modificando in parte quanto già presente, in particolare la classe `DBLink` dovrebbe essere estesa.
Riguardo al modulo Populator i due strati inferiori, Connector e Handler, potrebbero anche essere esportati ed utilizzati per altri progetti, perché si occupano di fornire un servizio del tutto generico allo strato superiore: connessione, creazione, inserimento. Lo strato più alto, Populator, invece, è facilmente estendibile con l'aggiunta di metodi specifici per il popolamento di eventuali nuove tabelle, ma è difficilmente riutilizzabile perché anche in questo caso il codice è specifico del problema affrontato.
Altre classi che potrebbero essere utilizzabili in altri progetti sono le classi di gestione dei PDB e delle specie, nonché quella di gestione dei parametri di accesso al database.

5.2.1 Esecuzione periodica del programma

Il programma è stato progettato per essere ri-utilizzato periodicamente al rilascio di nuove versioni dei database sorgenti. Sarà sufficiente sostituire i vecchi file con le nuove versioni, eseguire un `DROP` del database corrente e ri-eseguire l'intero programma da zero. Allo stato attuale non è prevista una versione del programma che permetta di effettuare l'aggiornamento del database con i nuovi dati.

5.3 File di configurazione del programma

Il programma può essere configurato attraverso due file di configurazione principali. Di default questi file sono il **config.cfg** e il **tables.cfg**. Entrambi questi file vengono passati da riga di comando in fase si esecuzione del programma, oppure vengono selezionati di default se non vengono passati argomenti.

5.3.1 Il file config.cfg

In questo file si trovano tutti i nomi dei file utilizzati dal programma, sia quelli originali dei quattro database, che quelli elaborati da Parser e Mapper. È inoltre codificata anche la struttura gerarchica dei directory dove andranno memorizzati i vari file. Il modulo Parser si occuperà, come prima cosa, di creare le cartelle (se non esistono già) e prelevare / creare tutti i file necessari.
Il file è strutturato secondo uno schema chiave-valore, la chiave viene letta dal programma e non deve essere modificata, a meno di modificare anche il codice sorgente, mentre si può modificare il valore, ad esempio per usare un diverso nome per i file, o una diversa root directory.

Per il corretto funzionamento del programma è necessario che venga creata manualmente la cartella **Original** (o equivalente) come specificata nel file di configurazione e che vi siano copiati dentro i quattro file sorgenti dei database, ovvero:

- file UniProt: *uniprot-all-reviewed*;
- file PTM: *dbPTM2.txt*;
- file ELM: *PhosphoELM-9.0_original.txt*;
- file HPRD: *POST_TRANSLATIONAL_MODIFICATIONS.txt*.

Inoltre nella cartella **Support** (o equivalente se è stata modificata nel file di configurazione) vanno inseriti i tre file:

- *DB_Parameters.txt*;
- *HPRD_ID_MAPPINGS.txt*;
- *databaseGenerator.sql*.

Il primo contiene i parametri necessari alla connessione al database, mentre il secondo contiene il mapping tra gli ID HPRD e gli AC e viene usato in fase di mapping tra i database.
Il terzo file contiene il codice sorgente usato dal programma per creare il nuovo database con tutte le relative tabelle.
Tutte le altre cartelle possono venire create direttamente dal programma.
Il file *DB_Parameters.txt* va passato da riga di comando e modificato (eventualmente) manualmente. In questo file si trovano anche nome utente e password di accesso al database.

5.3.2 Il file tables.cfg

La struttura di questo file è simile a quella del file *config.cfg*: i valori associati alle chiavi sono i nomi delle tabelle del database, in questo modo se si vuole modificare il nome delle tabelle in fase di creazione basta agire su questo file. Per il corretto funzionamento del programma, però, è necessario modificare manualmente i nomi anche nel file *databaseGenerator.sql* (si veda la sezione 6.4.2 del capitolo sugli sviluppi futuri per ulteriori dettagli).

Capitolo 6

Sviluppi futuri del progetto

In questo capitolo vengono presentati gli ambiti futuri di sviluppo del progetto tramite l'aggiunta di nuove funzionalità, la soluzione di alcuni problemi attualmente irrisolti e il miglioramento di alcuni aspetti.

6.1 Creazione di un front end web

Per rendere agevole l'utilizzo del database è prevista la futura creazione di un front end utilizzabile via web, ovvero un'interfaccia che permetta la consultazione semplificata dei dati presenti nel database. L'applicazione dovrà prevedere anche un meccanismo di query non predefinite nel caso si vogliano estrarre dati particolari. L'interfaccia dovrà permettere di visualizzare chiaramente le informazioni riguardanti le proteine e i siti di fosforilazione e dovrà consentire l'accesso ad informazioni analoghe a quelle prodotte dalle query di esempio (si veda la sezione 3.2 sulle query del capitolo sui database) in modo semplice, senza dover scrivere l'intera query SQL.

6.2 Problemi relativi alle chinasi

Le chinasi presenti nei quattro database sorgenti non vengono sempre identificate correttamente dal programma di parsing per due motivi:

1. può essere che i dati nei file sorgenti siano misti, irriconoscibili, contengano dei commenti oppure siano troncati. In questo caso i dati vengono salvati nei file "*X_unknown.txt*" dove X è il nome del database;

2. l'altra possibilità è che il nome della chinase sia chiaro, ma che il file sorgente non fornisca un modo sicuro per interpretare l'informazione come chinase o come gruppo di chinasi. In questa seconda situazione i dati vengono salvati nei file "*X_uncertain.txt*".

L'altro problema relativo alle chinasi è che, allo stato attuale, non è previsto un collegamento tra chinase e gruppo di chinasi di appartenenza.

6.2.1 Collegare le chinasi ai gruppi

Le singole chinasi possono appartenere ad un gruppo di chinasi più ampio. Questi collegamenti non sono deducibili con precisione utilizzando i soli file sorgenti dei quattro database e richiederanno l'uso di strumenti esterni (ad esempio un database online di mapping). Per creare il collegamento si dovranno modificare sia il codice sorgente del programma di popolamento, sia lo schema del database introducendo un'informazione sul gruppo di chinasi di appartenenza nella tabella delle chinasi oppure aggiungendo una nuova tabella di collegamento tra le due.

6.2.2 Approfondire il parsing

Come già accennato il parsing delle chinasi deve essere approfondito. Per quanto riguarda i file "uncertain" si può cercare di capire se una voce è una chinase o un gruppo analizzandone il nome e confrontandolo con quello dei gruppi noti, ma questa soluzione è imprecisa perchè può succedere che una chinase e un gruppo abbiano lo stesso nome. Un'alternativa è quella di utilizzare uno strumento esterno per risolvere l'ambiguità.
Il problema è, invece, più complesso per quanto riguarda i file "unknown" in quanto le informazioni sono molto eterogenee, alcuni esempi trovati nei file PTM e UniProt:

```
in form 4-P, form 5-P, form 6-P and form 7-P
by ATR; in vitro
by MAPK; in C3, C4, C5 and C6
by PHK; in form phosphorylase A (By similarity)
by host; in p58 (By similarity)
ACTIVATES THE KINASE
CK; IN ISOFORM HMG-IAND ISOFORM HMG-Y
```

6.3 Problema dello shift delle sequenze primarie

Un altro problema attualmente non risolto riguarda lo shift delle posizioni degli aminoacidi nelle sequenze primarie: può succedere che due database diversi contengano informazioni sulla stessa proteina, compresi eventualmente gli stessi siti di fosforilazione. Tuttavia può succedere che i due database considerino sequenze primarie apparentemente diverse per la proteina: in genere quando questo succede una delle sequenze è solo "shiftata" rispetto all'altra di qualche posizione, ma può anche succedere che le sequenze abbiano lunghezze diverse quando i dati di un database non sono precisi.

Poichè ci sono solo due database che memorizzano le sequenze primarie tra quelli utilizzati non è possibile adottare un criterio di "maggioranza" per determinare quale sia la sequenza primaria corretta. Quindi quando si verificano problemi di shifting, per il momento, le due versioni dei dati sulla proteina vengono scartati. Sarà necessario ricorrere all'uso di qualche database esterno per identificare questi problemi, oppure implementare qualche routine di string matching per tentare di "allineare" le sequenze.

È anche possibile che lo shift si verifichi all'interno dello stesso database nel caso di HPRD perchè quest'ultimo tiene traccia anche dell'*isoform* cui fanno riferimento i dati sui siti. Per esempio per la proteina avente nuovo ID = 529706 il database HPRD fornisce, per il sito in posizione 549 due diversi tipi di fosforilazione in base all'isoform considerato. È quindi necessario stabilire anche quale sia l'isoform "canonico" da utilizzare. Nell'esempio fatto, supponendo uno shift di un aminoacido, è possibile che la proteina possieda due siti di fosforilazione diversi in posizioni contigue che quindi si trovano a sovrapporsi creando una situazione doppiamente errata, in quanto non solo vengono scartati i due siti sovrapposti, ma vengono erroneamente memorizzati quelli che non si sovrappongono, come mostrato in figura.

Figura 6.1: Erroneo inserimento dei dati nel database dovuto allo shif.

Le aree colorate in rosso rappresentano un sito che alla stessa posizione presenta due diversi tipi di fosforilazione ed è quindi sbagliato, mentre le aree gialle rappresentano siti che verrebbero inseriti in quanto non sovrapposti ma che, evidentemente, sono comunque sbagliati, o almeno lo è uno dei due.

6.4 Problemi minori e miglioramenti

Il progetto contiene alcuni problemi minori che non ne influenzano gravemente il funzionamento ma che in futuro andrebbero risolti, inoltre è anche possibile migliorare e/o ottimizzare ulteriormente quanto già sviluppato.

6.4.1 Il problema dei fake ID

Come già accennato precedentemente (si veda l'interfaccia `Parser` - sezione 4.3.6) nel database ELM non sono presenti gli ID tassonomici delle specie cui fanno riferimento le proteine e in alcuni casi queste specie non sono presenti in UniProt, ciò porta alla creazione di alcune voci di specie aventi un ID tassonomico minore di -1 che ovviamente non ha una corrispondenza nel database NCBI. Uno sviluppo futuro del progetto dovrà prevedere un modo per eliminare questi ID artificiosi cercando quelli reali. Ciò può essere fatto, ad esempio, attraverso delle query al database NCBI.

6.4.2 Parametrizzare i nomi delle tabelle del database

Uno sviluppo non essenziale ma molto utile sarebbe quello di parametrizzare il nome delle tabelle nel file di generazione del nuovo database. Attualmente se si vogliono modificare i nomi delle tabelle bisogna modificare due file distinti: il file "databaseGenerator.sql" e il file "tables.cfg". Questo comportamento può essere migliorato modificando il metodo di creazione del database (che legge il file databaseGenerator.sql) in modo da utilizzare per i nomi delle tabelle i valori contenuti nel file tables.cfg. Per ottenere questo risultato occore che nel file con il sorgente SQL, al posto dei nomi delle tabelle, vengano inserite le chiavi usate nel file tables.cfg.