知能プログラミング演習II 課題2

グループ8
29114003 青山周平
29114060 後藤拓也
29114116 増田大輝
29114142 湯浅範子
29119016 小中祐希

2019年10月25日

提出物　rep2，group08.zip

1　課題の説明

課題 2-1 Matching クラスまたは Unify クラスを用い, パターンで検索可能な簡単なデータベースを作成せよ.

課題 2-2 自分たちの興味ある分野の知識についてデータセットを作り, 上記 2-1 で実装したデータベースに登録せよ. また, 検索実行例を示せ. どのような方法でデータセットを登録しても構わない.

課題 2-3 上記システムの GUI を作成せよ.
・データの追加，検索，削除を GUI で操作できるようにすること.
・登録されたデータが次回起動時に消えないよう, 登録されたデータをファイルへ書き込んだり読み込んだりできるようにすること.

2　必須課題 2-1

Matching クラスまたは Unify クラスを用い，パターンで検索可能な簡単なデータベースを作成せよ．

私の担当箇所は，データセットからデータを読み取り，データベースに格納するためのメソッドの作成である．

2.1　手法

課題内容の実装のために，以下のように改良を加えた．

1. 与えられたデータセットを DB に格納する．
2. DB にアクセスするための DAO を作成する．
3. DB から得られたデータと，検索文を比較して検索を行えるよう Matching.java と Unify.java を改良する．

1. に関して，テキストファイルと DB 間でデータのやり取りを行うため，テキストファイルにアクセスするための java ファイル TextCon.java を作成した．そして 1. の実現のため，テキストファイルからデータを読み込むためのメソッド readTextFile を作成した．これは私が担当した．

2. に関しては，DB アクセスを行う際のデザインパターンである DAO の作成のため，TextDAO.java を作成した．具体的には table(texts) を作成し，column として uuid(int 型) と line(text 型) を定め，そこにテキストファイルの中身が入るようにした．これも私が担当した．

3. に関しては，与えられた Unify.java では変数入力の際に値を返すことが出来ないため，変数検索にも対応可能になるようプログラムの書き換えを行った．これは後藤君と小中君が担当した．

2.2　実装

まずテキストファイルから DB を利用するためのプログラムを作成した．始めに DB とその table を作成し，その内容を初期化する．その後与えられた「テキストファイルに対する操作のプログラム」を利用してテキストファイルから一文ずつ取り出し DB に格納する．

このとき，テキストファイルを読み込むためのメソッドは readTextFile，DB とその table を作成するメソッドが createTab，DB の table 内部を初期化するメソッドが deleteData，読み込んだデータを DB に格納するメソッドが insertTextTab である．

テキストファイルアクセスと DB アクセスを分けるため，それぞれ TextCon.java，TextDAO.java というファイルを作成し，readTextFile メソッドは TextCon.java に，createTab，deleteData，insertTextTab メソッドは TextDAO.java に含まれている．

また，DB アクセスプログラムは，3 年生前期の授業で取ったプログラミング応用の課題で作成した DAO などのプログラムも参考にして作成を行った．

テキストファイルを読み込んで DB アクセスを指示する readTextFile メソッドの実装をソースコード 1 に示す．

ソースコード 1: readTextFile メソッド

```
1  public void readTextFile() throws FileNotFoundException{
2          String empty = "";
3          //text ファイル処理
4          try {
5                  // table の作成【DB】
6                  TextDAO.createTab();
7                  TextDAO.conCom();
8                  // table の中身の初期化【DB】
9                  TextDAO.deleteData();
10
11                 // ファイル読み込み操作
12                         …
13                         if (!line.equals(empty)) {
14                                 // 更新【DB】
15                                 TextDAO.insertTextTab(line
                                       );
16                         }
17                         //【DB】
18                         TextDAO.conCom();
19                 }
20                 in.close(); // ファイルを閉じる
21         } catch (IOException e) {
22                 e.printStackTrace();
```

```
23          }
24          //【DB】
25          TextDAO.conCom();
26          TextDAO.closeConn();
27  }
```

DB の table 作成のための createTab メソッドをソースコード 2 に示す.

ソースコード 2: createTab メソッド

```
1  public static Connection conn = null;
2  public static final String connDB = "jdbc:sqlite:data.db
       ";
3  // 格納用table を作成
4  public static void createTab(){
5          PreparedStatement pStmt = null;
6          String sql;
7          try{
8                  Class.forName("org.sqlite.JDBC");
9                  conn = DriverManager.getConnection(connDB
                       );
10                 conn.setAutoCommit(false);
11                 sql = "create table texts(uuid int, line
                       text)";
12                 pStmt = conn.prepareStatement(sql);
13                 pStmt.executeUpdate();
14         }catch{
15                 …
16         }
17 }
```

DB の table の中身の初期化のための deleteData メソッドをソースコード 3 に示す.

ソースコード 3: deleteData メソッド

```
1  // DBtable の中身の削除
2  public static void deleteData() {
3          conn = null;
4          PreparedStatement pStmt = null;
5          try{
6              Class.forName("org.sqlite.JDBC");
7              if(conn == null){
8                          conn = DriverManager.getConnection
                               (connDB);
```

```
9                       conn.setAutoCommit(false);
10              }
11      String sql = "delete from texts";
12      pStmt = conn.prepareStatement(sql);
13              pStmt.executeUpdate();
14      }catch{
15              …
16      }
17 }
```

DB にデータを格納するための insertTextTab メソッドをソースコード 4 に示す.

ソースコード 4: insertTextTab メソッド

```
1  // table に追加
2  public static void insertTextTab(String line) {
3          PreparedStatement pStmt = null;
4          String sql;
5          try{
6                  Class.forName("org.sqlite.JDBC");
7                  if(conn == null){
8                          openConn();
9                  }
10                 sql = "insert into texts values(?, ?)";
11                 pStmt = conn.prepareStatement(sql);
12                 pStmt.setInt(1, id);
13                 pStmt.setString(2, line);
14                 pStmt.executeUpdate();
15                 id++;
16         }catch{
17                 …
18         }
19 }
```

またこれらの動作のため，DB へのコネクション OPEN として openConn メソッド，CLOSE として closeConn メソッド，COMMIT として conCom メソッドを新たに作成している．これらは TextDAO.java と TextCon.java のどちらからも利用される．

2.3 実行例

作成したDBへの格納プログラムが正しく動作しているかを確認するためにTest.javaを作成し，readTextFileメソッドを実行した．すると，data.dbという名前のDBが作成される．この中身を確認したところ，以下のように表示された．

```
C:\Users\Owner>sqlite3 data.db
SQLite version 3.28.0 2019-04-16 19:49:53
Enter ".help" for usage hints.
sqlite> .table
texts
sqlite> select * from texts;
1|Hanako is a girl
2|Hanako is a student
3|student is a kind of human
4|human is a kind of mammal
5|Hanako has a hobby of playing video-games
6|Hanako has a hobby of playing air-guitar
7|Hanako studies philosophy
8|Hanako loves Taro
9|Taro is a boy
10|Taro is a student
11|Taro has a hobby of playing video-games
12|Taro studies informatics
13|Taro loves Jiro
14|Taro has a pet named Jiro
15|Jiro is a boy
16|Jiro is a dog
17|dog is a kind of mammal
18|Jiro has a hobby of playing frisbee
19|Jiro loves Hanako
sqlite>
```

この結果はテキストファイルに含まれていたデータセットの内容と一致することから，作成したプログラムが正しく動作し，DBが作成できていることが確認できた．

2.4 考察

この課題は課題文の解釈の方法がいくつか考えられたため，どのようにDBを作成するかを考えるのにかなりの時間を要した．

まず，DBの作成を行うため，データセットであるテキストファイルからどのようにDBを作成するかを考えた．この時大きく分けて2種類の作成方法がまず考えられた．それを以下に示す．

1. DBにデータセットと同様の内容を格納する．
2. DBにUnifyで処理した検索文(変数を含む文章)とその出力結果を格納する．

この2種類の考えは，課題文の解釈の違いによって考えられた．

「MatchingクラスまたはUnifyクラスを用い，パターンで検索可能な簡単なデータベースを作成せよ.」という課題であるが，Matchingクラスや Unifyクラスをどの段階で使用するのかが明確に判断できず，DB作成後にMatchingクラスやUnifyクラスを利用するのか，Matchingクラスや Unifyクラスを利用して得られた結果をDBに格納するのかが分からなかったからである．Matchingクラスや Unifyクラスを使用する前にDBを作成する場合は1. のDBの形式にし，Matchingクラスや Unifyクラスを使用した後にDBを作成する場合は2. のDBの形式にすると考えられたが，どちらを採用するかがかなり難しい判断であった．

さらに，DBをデータセットと同様の内容とした場合はさらに3種類の作成方法が考えられた．それを以下に示す．

1. DBに一文をそのままtext型として格納する．
2. DBに一文を各単語毎に分割して格納する．
3. DBで動詞毎にtableを作成し，それぞれの動詞で分類分けした状態で主語とそれ以外(動詞も除く)に分割して格納する．

DBをより効率よく利用する方法を考え，これらのような格納方法を考えた．

DB の作成方法に複数の方法が考えられたが，これは必須課題の 1 つ目であり，作成期間も一週間と短く，出来るだけ早く方針を固めなければならない．そこで私たちの班では，DB のこれ以降の課題との関係性や検索方法と課題文の内容にどれだけ沿うことが出来るかなどを考慮してプログラムを作成することとした．

ここで，先に挙げた DB の格納方法とその具体例を表形式にまとめ，それぞれの利点と欠点を上げていく．

1 つ目は「DB に一文をそのまま text 型として格納する.」方法である．

表 1: DB 例 1(table:texts)

uuid(int)	line(text)
1	Hanako is a girl
2	Hanako is a student
···	···
8	Hanako loves Taro

表 1 の利点は，DB への格納が単純であるため比較的楽にプログラムすることが出来る点にある．また，Unify.java 等がデータを受け取るときにテキストファイルを読み込む形と殆ど変わらずにプログラム出来るため，改良の手間が少し減る点もあげられる．

これに対し欠点は，格納方法がテキストファイルと同様の形式であるため，DB へ格納した恩恵をあまり得られない点にある．データの削除や追加はテキストファイルよりも楽に行うことが出来るが，検索に関しての手間はテキストファイルと変わらないため，あまり意味のない DB になってしまう可能性も考えられる．

2 つ目は「DB に一文を各単語毎に分割して格納する.」方法である．

表 2 の利点は，検索が DB を用いて行うことが出来るために，マッチング操作にかかる手間が大幅に減ることがあげられる．

これに対し欠点は，一文に含まれている単語の数が一定でないため，上の表のようにフィールドによっては NULL になってしまい DB への格納が複雑になることがあげられる．一文が分割される単語数もテキストファ

表 2: DB 例 2(table:texts)

uuid(int)	namea(text)	nameb(text)	namec(text)	named(text)
1	Hanako	is	a	girl
2	Hanako	is	a	student
…	…	…	…	…
8	Hanako	loves	Taro	NULL

イルに含まれる行数も可変であるため，可変長二次元配列を活用して格納を行わなければならなくなってしまい，かなり複雑なプログラムとなることが考えられる．またこの格納方法では，検索時に DB 参照によって検索結果を得られるため，使用するように指定されている Matching クラスや Unify クラスが利用できなくなってしまうことも欠点としてあげられる．

3 つ目は「DB で動詞毎に table を作成し，それぞれの動詞で分類分けした状態で主語とそれ以外 (動詞も除く) に分割して格納する．」方法である．

表 3: DB 例 3(table:iss)

uuid(int)	namea(text)	nameb(text)
1	Hanako	a girl
2	Hanako	a student
…	…	…

表 4: DB 例 3(table:lovess)

uuid(int)	namea(text)	nameb(text)
8	Hanako	Taro
…	…	…

表 3, 表 4 の利点は，table を複数作成することで，テキストファイルに

は出来ない単語毎の分類が可能になるので，DB を利用する利点が明確になることである．同時に，分類分けが行われることで検索時のマッチング操作にかかる手間が減ることも考えられる．

これに対し欠点は，分割を行って動詞毎に table を作成するため，DAO が非常に複雑になってしまうことがあげられる．実際にこのプログラムは始めに作成したが，デバッグ作業にかなりの時間を要することになった．また各 table 毎に検索を行うため，Unify.java に引数としてデータを与える際に工夫が必要になることも，実行効率は良いが実装効率が悪いことを示している．さらに調べたところ，table として指定できない特定の文字列が存在し，それが is が該当していると分かった．そのため動詞毎で table を作成する場合は全ての動詞に's' を付けるなどの工夫が必要になることが課題を解く中で分かった．これも，プログラムが複雑になる原因の一つとして考えられる．

最後に,「DB に Unify で処理した検索文 (変数を含む文章) とその出力結果を格納する.」方法である．

表 5: DB 例 4(table:answers)

uuid(int)	question(text)	answer(text)
1	?x has a hobby of playing video-games	Hanako, Taro
2	Hanako is a ?y	girl
…	…	…
8	?x is a boy, ?x loves ?y	(Taro, Jiro), (Jiro, Hanako)

表 5 の利点は，必須課題で求めらているプログラムに最も近いと考えられるプログラムであることがあげられる．また，データセットと DB の役割の違いがはっきりするため，テキストファイルと DB を有効に活用できている部分も利点と考えられる．

これに対し欠点は，格納のためのプログラムが複雑になってしまうことと，発展課題を行う際に追加と削除のデータが DB に反映させられないことがあげられる．DB を利用することで，テキストファイルよりも追加や削除が簡単になるが，この方法で実装を行うとこれが出来ない．それにより追加や削除命令に応じてテキストファイルを直接操作することになるが，これは非常に手間がかかり効率も悪い．

これらの利点と欠点，そして実装期間を考え，私たちの班は表 1 の方法を採用することとした．これらを踏まえ作成したものが先に述べたプログラムである．そのため，結果として DB はテキストファイルと形式を殆ど変えることなく作成することとなった．ただしこのとき，改行は DB に含まないことにしたため，DB では空行は含まれずに格納される．

これにより実装を分担して行う今回のような課題は実装方法が複雑にならなかったことから大幅に時間をかけることなくプログラムを作成することが出来た．

3　課題 2-1(-1)

与えられたパターンにマッチする全データを列挙するプログラムを作成せよ.
例えば, この例のような形式のデータセットから,?x has a hobby of playing video-games や Hanako is a ?y のような, 様々なパターンにマッチするデータを検索できるようにすること.

3.1　手法

Matching.java でのマッチングの大まかな手順は次のようになる.
また, 今回はサンプルプログラムとして用意されている Matching.java を必要に応じて書き換えることによりマッチングを実現した.

1. Matching クラスを定義し,Matcher クラスのインスタンスを生成する
2. 引数にとった二つの文章が等しいかどうか比較する
3. 文章をトークンごとに分割し, トークンの数を比較する
4. トークンごとの比較をする
5. トークンの片側に変数が含まれていたらハッシュに登録されているかを確認し, 未登録であれば格納する
6. 他のトークンが全てマッチング成功したら, 変数に対応するトークンを出力する

3.2　実装

今回用意されていた Matching.java にはマッチングに必要なメソッド自体はすでに含まれていたが, 返り値が全て boolean であったため, マッチングの結果を出力できるように書き換えた.

まず, プレゼンターとのやり取りを行うために main メソッド内に必要な要素を追加した.

また, テキストの読み込み等はプレゼンター側で行うため削除し, テスト実行用として出力文を追加している.(ソースコード 1)

ソースコード 5: main メソッドの変更

```
List<String> answer = new ArrayList<String>(); // 答えを格納

        View view = new View(); // インスタンス?の作成
        Presenter presenter = new Presenter(view);

        presenter.start(); // DB の作成

        presenter.searchData(arg[0]); // 検索語入力
        answer = view.getSr(); // 答えを取得

        //answer.add((new Matcher()).matching(arg[0],line));//ここでarg[0]引数 1, arg[1]に引数 2

        //テスト出力用
        for(int i = 0; i < answer.size(); i++) {
                if(answer.get(i) != " ") {
                        System.out.println("★answer = " + answer.get(i));
                }

        }
```

次に, マッチングした結果を実際に出力できるようにするための変更点を示す.
Matcher クラスや Matcher クラスのインスタンスを定義する部分では, 新しく変数 flag やリスト KeyList を追加している.(ソースコード 2)

ソースコード 6: 追加要素

```
class Matcher {
          StringTokenizer st1;
          StringTokenizer st2;
          HashMap<String,String> vars;
          ArrayList<String> KeyList; //Key をとり, HashMap の Value を取るためのリスト
          int flag; //ミスフラグ

          Matcher(){
```

```
9                   vars = new HashMap<String,String>();
10                  KeyList = new ArrayList<String>();
11                  flag = 0;
12          }
```

matching メソッドでは,boolean ではなくマッチング結果（String）を返したいので, メソッドを String 型に変更した.
マッチングが成功した場合は該当のトークンを, 出力がない（完全一致, マッチングに失敗など）場合は空白を返している.（ソースコード 3）

ソースコード 7: matching メソッド

```
1  public String matching(String string1,String string2){
2                   //System.out.println("tagetData = " +
                        string1);
3                   //System.out.println("resultList = " +
                        string2);
4
5                   // 同じなら成功
6                   if(string1.equals(string2)) return " ";
7
8                   // 各々トークンに分ける
9                   st1 = new StringTokenizer(string1);
10                  st2 = new StringTokenizer(string2);
11
12                  //System.out.println("トークンの数" + st2.
                        countTokens());
13
14                  // 数が異なったら失敗
15                  if(st1.countTokens() != st2.countTokens())
                         return " ";
16
17                  // 定数同士
18                  for(int i = 0 ; i < st1.countTokens();){
19                      if(!tokenMatching(st1.nextToken(),st2.
                            nextToken())){
20                          // トークンが一つでもマッチングに失
                                敗したら失敗
21                          return " ";
22                      }
23                  }
24
25
```

```
26                  // 最後まで O.K. なら成功
27                  //System.out.println("成功! " + vars.get(
                        KeyList.get(0)));
28                  return vars.get(KeyList.get(0));
29          }
```

varMatching メソッドでは, 初めてトークンが var に追加される際にリスト KeyList に vartoken を追加することでマッチングの答えを取得, 保持している. (ソースコード 4)

ソースコード 8: varMatching メソッド

```
1  public boolean varMatching(String vartoken,String token){
2                   if(vars.containsKey(vartoken)){
3                       if(token.equals(vars.get(vartoken))){
4                           //System.out.println(token);
5                           return true;
6                       }
7                       else {
8                           return false;
9                       }
10                  } else {
11                      vars.put(vartoken,token);
12                      KeyList.add(vartoken);
13                      //System.out.println(vars.get(vartoken
                            ));
14                  }
15                  return true;
16          }
```

3.3 実行例

今回は以下のデータセットを対象に, マッチングのみを行った例を示す.

ソースコード 9: データセット

```
1  Hanako is a girl
2  Hanako is a student
3  student is a kind of human
4  human is a kind of mammal
5  Hanako has a hobby of playing video-games
6  Hanako has a hobby of playing air-guitar
```

```
7 Hanako studies philosophy
8 Hanako loves Taro
9 Taro is a boy
10 Taro is a student
11 Taro has a hobby of playing video-games
12 Taro studies informatics
13 Taro loves Jiro
14 Taro has a pet named Jiro
15 Jiro is a boy
16 Jiro is a dog
17 dog is a kind of mammal
18 Jiro has a hobby of playing frisbee
19 Shuhei is a boy
```

引数には以下の質問文を与えた.
”?x is a boy”

実行結果は以下のようになった.

ソースコード 10: 実行結果

```
1 Successfully started
2 targetData1 = ?x is a boy
3 検索結果を取得
4 ★answer = Taro
5 ★answer = Jiro
6 ★answer = Shuhei
```

3.4 考察

今回の課題ではサンプルプログラム Matching.java があらかじめ用意されていたので, ここではサンプルプログラムをハンドトレースしていって理解するまでのプロセスにおいて感じたこと, 考えたことを記す.
まずプログラムを見る前にイメージした手法だが, 本課題において求められているパターン照合は単語単位でのマッチングが必要になってくるので, 単に String 型の文章同士を比較するのでは駄目だということが直感的にわかる.
ウェブインテリジェンスを履修していたので形態素解析が思いついたが, 今回は英文なので単語間に空白があり, 実際はそのことを利用して String-Tokenizer で分割をしている. 日本語などにも対応させようとすると形態

素解析などの作業が必要になってくるのではないだろうか.
次に matching だが、トークンごとに比較する時に単に一致したら成功というだけでなく, 片側が変数で片側が定数という場合を例外として扱わなければならない. 今回のプログラムでは複数の boolean 型のメソッドをうまく用いて課題通りのマッチングを実現しているが, 返り値が boolean なために実際に求めた単語を素直に返す, ということができなかった. ここが今回の 2 − 1 において一番やっかいな部分であったと感じた.
今回はメソッドの外に共通のリストを用意し,varMatching の段階で格納, 保持することで実現することができた.

4 課題 2-1(-2)

> 複数のパターンが与えられたときに全ての可能な変数束縛の集合を返すようなプログラムを作成せよ．例えば，上記の例で「?x is a boy」と「?x loves ?y」の両方が与えられたときに，(?x, ?y) の全ての可能な変数束縛の集合として (Taro, Jiro), (Jiro, Hanako) を返すこと.

4.1 手法

課題を達成する (つまり, 「?x is a boy」と「?x loves ?y」と引数を取ると, (Taro, Jiro), (Jiro, Hanako) を返す) ために以下のようなプログラムの処理を行う.

1. 全体をまとめるハッシュマップを 1 つ生成する.
2. 1 つ目の引数, 2 つ目の引数を順番に処理を行う.
3. text ファイルからデータセットを 1 行ずつ読み取り, 1 行ずつ引数と照合する.
4. 1 行はトークン単位に分割され, マッチングは各トークンごとで行う.
5. 対応した場合, ハッシュマップに Key として変数 (?x や?y など) を, Value として具体定数 (Taro や Jiro など) を加えていく.

6. 1 つ目の引数で束縛された変数は, 2 つ目の引数に反映される (1 つ目で?x=Taro と決まれば, 2 つ目の引数ではその条件を引き継がせる).

7. ハッシュマップを 2 次元リストにし, 中身 (Key や各 Key に対応する Vlaue) をリスト化することで, 全てのマッチング可能なパターンを 1 つのハッシュマップに格納できる.

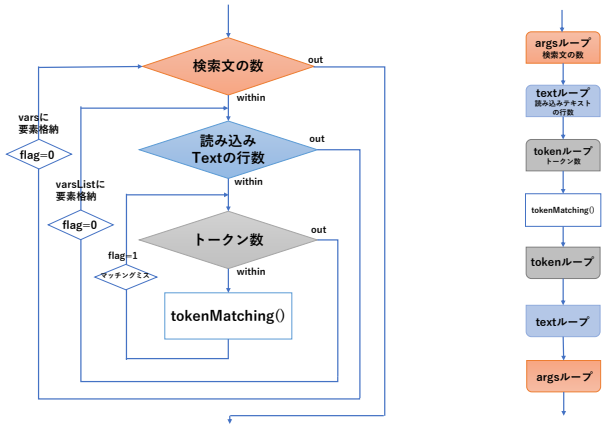

図 1: unify プログラムの 3 つのループ

プログラムの流れは多段階のループ構造を構築している. 図 1 における”3 つの大きなループ”を参照し, 上記の 7 つの手法について詳しく説明する.

1. に関して, main クラスで 1 度 Unifier クラスのインスタンスが生成されるとコンストラクタにより, ハッシュマップをはじめとするあらかじめ宣言されていた複数の変数やリストを初期化する. ここで初期化され生成されたインスタンスは, 今後のプログラム全体で使われる.

2. に関しては, 2 つの引数をリストに格納させ, ループ処理を行う. ここが 1 つ目の大きなループである.

3. に関しては, BufferedReader オブジェクトによりテキストを 1 行ずつ読み込む. ここが 2 つ目の大きなループである.

4. に関しては, 既存のプログラムを利用し, StringTokenizer により分解したトークンの数だけループ処理をする. ここが 3 つ目の大きなループである. 引数と読みこんだ 1 行とでトークン数の異なる場合は, トークン数の多いほうにそろえるように, 少ないほうに null を入れて数を調整する.

5. に関しては, トークン単位のマッチングは成功しても, 1 行を通して成功していなければ成功とはいえない. 例えば, 「?x is a boy」と「Hanako is girl」は, トークン単位のループ処理ででマッチングをしていくと, 先に「?x = Hanako」でマッチングが成功するが, その後「boy != girl」となるので, 今回は失敗であり, ハッシュマップには追加できない. これを実現するために, 1 行すべてのマッチングが成功した後に, これまで成功していたトークンの組をハッシュマップに保存する処理を行う. 具体的には, マッチングが成功したトークンの組はそれぞれ一時バッファならぬ一時リストに追加して, ハッシュマップには追加しない. マッチングの成功不成功を判断する”ミスフラグ”を用意し, 1 行におけるトークンの組み合わせすべてが成功したら (ミスフラグが上がらなかったら), 先ほどの一時リストの要素をハッシュマップに保存する.

6. に関しては, 「?x = Taro, Jiro」と 1 つの変数において, 2 つの定数が束縛されることがあるので, それぞれにおいて 2 つ目の引数を変更する. 具体的には, 2 つ目の引数は「?x loves ?y」から「Taro loves ?y」と「Jiro loves ?y」の 2 つに変更させる. そのため, 手法 1 説明したで 1 つ目の大きなループは, 引数 2 つであっても, 3 回ループが行われる.

7. に関しては, 上記の手法 6 で述べたように, 1 つの Key(たとえば?x) に対して, 複数の Value 値 Taro, Jiro を持つので, リスト化する必要があった. 上記の手法 6 で”トークン (語) の単位で行うのではなく, 1 行単位でハッシュマップに保存をする”と述べたが, この際, 実際のマップに保存するわけではなく, 2 次元リストであるハッシュマップに格納するためのリストに保存している.

4.2 実装

上記の 7 つの手法の中で, 特に重点を置いた箇所に関して, プログラムを参照にしつつ説明していく.

手法 5 に関する「1 行を通してすべてのトークンがマッチングに成功した時のみ, ハッシュマップに反映させる」の部分を実装したソースコード 11 に示す。

ソースコード 11: 1 文すべて終わったら格納する

```
1
2   for(int i = 0 ; i < length ; i++){ //1語ずつマッチング
        していきます
```

```
3         System.out.println("flag = " + flag);
4
5         if(!tokenMatching(buffer1[i],buffer2[i])){ //マッ
              チングできてないなら...
6                 System.out.println("Search Error 2");
7
8                 flag = 1; //miss フラグ上げる
9         }
10
11    } //1文の解析がすべて終わって...
12    if(flag == 0) { //missflag が一度も上がっていないなら,
13          for(int len = 0; len < ValueList.size(); len++) {
14              //HashMap に格納せずに,
15              //vars.put(KeyList.get(len), ValueList.get(len
                    ));
16              System.out.println("ValueList.get("+ len + ")
                     ="+ValueList.get(len));
17
18              //varslist に格納
19              varslist.add(ValueList.get(len));
20          }
21    } //Text の文全てが終わったら...
```

flag はフィールド変数として, Unifier クラスのどのメソッドでも用いられる. 上記には書かれていないが, tokenMatching メソッドの varMatching メソッド内で 1 単語 (トークン) をマッチングさせるたびに, マッチング成功の成否に合わせ flag 管理をしている.

また, ハッシュマップは実際には 2 次元リストにより構築されているため, 1 文解析がすべて終わり, フラグが立ってない場合の処理だが, 手法 5 の説明時には簡略化のために省略したが, 実際には”ハッシュマップに格納する”のではなく, ”ハッシュマップに格納するためのリスト varslist に登録する”である.

手法 6 に関する「1 つ目の引数により得られた束縛条件を 2 つ目の引数に反映する」の部分を実装したソースコード 12 に示す。

ソースコード 12: 第 1 引数から第 2 引数への変数束縛条件の引継ぎ

```
1  //1つ目の引数,2つ目の引数,順番に処理する!
2  for(int num = 0; num < args.size(); num++) {
```

```
3
4    if(num == 1) { //2回目で
5       //変数 (?x,?y などなど)の数だけ...
6       for(int keyNum = 0; keyNum < KeyList.size(); keyNum
            ++) {
7          //その変数に対応する値 (Taro, Jiro などなど)の数だ
               け
8          for(int valueNum = 0; valueNum < varslist.size
               (); valueNum++) {
9
10                //文字列の中に?が入っていたら...
11                if(string2.contains("?")) {
12                        //文字列として置き換えて,新しく作成
                             !
13                        String stringx = string2.replace(
                             KeyList.get(0), varslist.get(
                             valueNum));
14                        System.out.println("stringx = " +
                             stringx);
15                        args.add(stringx); //などなど
16                }
17           }
18        }
19        args.remove(1); //もとを置き換え
20        System.out.println("args.size() = " + args.size
             ());
21     }
22
23     varslist = new ArrayList<String>(); //複数人対応の
          Value:値を格納させる.
```

1 つ目のマッチングの結果で得られた Key を格納した KeyList, 各 Key に対応した Value を格納した varslist が用いられている. 具体的には, 1 つ目の引数で”?x is a boy”としたとき, ?x = Taro, Jiro が当てはまるため, KeyList には「?x」が, varslist には「Taro, Jiro」が含まれている. これを replace メソッドを用いて, 実際に「?x を Taro」という風におきかえている.

ポイントとしては, 引数は 2 つでも, 1 つ目の引数のマッチングの結果, 「?x = Taro, Jiro」のように複数対応する場合, 2 つ目の引数が「?x loves ?y」の際, ?x が 2 つ代入され, 「Taro loves ?y」と「Jiro loves ?y」と増えることである. そのため, 元あった「?x loves ?y」を消すために. remove

メソッドで代入前の 1 文を消している.

手法 7 に関する「ハッシュマップの 2 次元リスト化による要素代入」の部分を実装したソースコード 13 に示す。

ソースコード 13: ハッシュマップの 2 次元リスト化

```
1   if(flag == 0) {
2
3      //いま見てるKey の番号にしないと！
4        int index = KeyList.indexOf(String.valueOf(keyList
             .get(0)));
5        System.out.println("KeyList = " + KeyList.get(
             index));
6
7        /* 参照渡しだから,作ったリストを削除してももとの場
             所をささない.*/
8        ArrayList<String> array = new ArrayList<>();
9        if(vars.containsKey(KeyList.get(index))) { //すで
             に作ったことがあったら,
10           System.out.println("削除します");
11           array.clear(); //消す
12           array = arrayCopy;//コピーを戻す
13                  flag2 = 1;
14       }
15       arrayCopy = array; //コピーを取っておいて,
16       array.add(ValueList.get(0));
17       if(flag2 == 0)
18            vars2list.add(array);
19       System.out.println("vars2list() = " + vars2list.
             toString());
20       vars.put(KeyList.get(index), vars2list.get(index
             )); //改良HashMap に格納
21
22       flag2 = 0;
23  }
24   System.out.println("途中結果は" + vars.toString() +
         "\n");
25   flag = 0; //falg のリセット
```

ここの処理は Text の全ての行が読み込めたら行われる. 2 次元リスト

を作るためには, リストに格納する処理を行うが, その際, Key の数 (?x だけなら 1 つ, ?x と?y なら 2 つなど）に応じて, 複製する Value のリストの数も変わってくる. リストの名前には, 配列の buffer[i] のように数字を名前に付け加えることができないため, ループ処理を行うこのプログラム中では, とりあえず, リストを作成するが, すでにある Key に対してリストが存在していれば, ダブってしまうので, 削除する. このとき, java ではリストのオブジェクトも参照渡しであるので, ただ消すだけでは, 参照先を前のリストに戻すことができない. そのために, 前のリストをコピーしておき, 消した際には, コピーもとを参照させる.

ソースコード 14: 複数候補に対応した varMatching メソッド

```
1  boolean varMatching(String vartoken,String token){
2    /* (注意)
3      * すでに?x=Taro という制約がある状態で,さらに?x=
           Jiro を付け加えないといけない. うやむやにやっても,かぶ
           っている候補を増やしてしまうだけである. */
4   //すでにあるKey に対して...
5      //HashMap に vartoken というキー(Not 値)が存在するかどう
           か
6      if(vars.containsKey(vartoken)){
7            System.out.println("varslist.size() = " +
                varslist.size());
8
9            //まだvarslist には入ってないけど,
                Maticng 成功した場合,
10           if(varslist.size() == 0) {
11                  ValueList.add(token);
12                  keyList.add(vartoken);
13           }
14           //普段はこっちだよね
15           else {
16                 for(int i = 0; i < varslist.size(); i++)
                       {
17                     //すでに登録されている関係なら...
18                     if(token.equals((vars.get(vartoken)).
                           get(i)))
19                        System.out.println("格納済み");
```

```
20                      else { //別の値(Taroじゃなくて Jiro)が
                            来たら, 加えます.
21                          ValueList.add(token);
22                          keyList.add(vartoken); //
                                Keyも含めて再登録しないと...
23                      }
24                  }
25          }
26          return true;
27      }
28
29    //初めてのKeyに関して...
30      else {
31          replaceBuffer(vartoken,token);
32          //HashMapの値リスト:
                varslistに含まれているかどうかで見る
33          if(varslist.contains(vartoken)) {
34              replaceBindings(vartoken,token);
35          }
36          //vars.put(vartoken,token); //ここではハッシュマ
                ップに登録しません.
37          if(!KeyList.contains(vartoken)) { //1回だけでっ
                せ!
38                  KeyList.add(vartoken);
39          }
40          ValueList.add(token);
41          keyList.add(vartoken);
42      }
43      return true;
44  }
```

上記は, 実際に 1(トークン) 単語単位でマッチングを行う varMatching メソッドである. ValueList と keyList は, テキストの 1 行ずつで初期化されるようになっており, ここのメソッド内でしかこの 2 つのリストは操作されない. この 1 行単位で初期化されるリストの結果は, 全体に反映されるリスト varslist に別のメソッドで格納されるので, それをもとに if 文の条件分岐を行っている.

4.3 実行例

引数に [”?x is a boy, ?x loves ?y”] としたときの実行結果が以下のようになる.

```
1 Successfully started
2 検索結果を取得
3 ★answer = ?x0 = Taro
4 ★answer = ?y0 = Jiro
5 ★answer = ?x1 = Jiro
6 ★answer = ?y1 = Hanako
```

?x には, Taro, Jiro が入り, その後, 2 つ目の引数は「Taro loves ?y」, 「Jiro loves ?y」と置き換えられ, ?y は Jiro, Hanako が入る. 出力結果を, それぞれの x と y に対応させるために, ハッシュマップから, Key と Value の for 文を回している.

正しい関係性が出力されていることが確認される.

同様に, [”?x is a student, ?x studies ?y”] と実行すると,

```
1 Successfully started
2 検索結果を取得
3 ★answer = ?x0 = Hanako
4 ★answer = ?y0 = philosophy
5 ★answer = ?x1 = Taro
6 ★answer = ?y1 = informatics
```

と出力されることが確認された.

4.4 考察

6 に関して, 1 つ目の引数で束縛された変数の値を, 実際に 2 つ目の引数に代入するので, もし「?x loves ?y」と「?x is a boy」という順番の引数の組み合わせだと最終的に間違ったハッシュマップが生じる. というのも, 「?x loves ?y」にあたる?x は [Hanako, Taro, Jiro] であるので, ハッシュマップその組み合わせが登録されるが, 「Hanako is a girl」なので, ミスが生じるが既にハッシュマップに登録されているので間違った出力となる. これは unify していない. ただの Matching である.... 引数も 2 つに限定させてしまった.

5　必須課題 2-2

自分たちの興味ある分野の知識についてデータセットを作り，上記 2-1 で実装したデータベースに登録せよ．また，検索実行例を示せ．どのような方法でデータセットを登録しても構わない．

必須課題 2-2 は実装を伴わない課題であるため，実装以外を記述する．

5.1　手法

私たちの班では，必須課題 2-1 で DB を表 1 のように作成したため，与えられたテキストファイルと同様の形式でデータセットを作成した．この課題は私が担当した．

興味のある分野については，ラグビーワールドカップが日本で開催され日本チームが活躍していたことから，ラグビーについて調べ以下のようにデータセットを作成した．

5.2　実行例

まず，必須課題 2-1 と同じ要領で作成したデータセットから DB を作成した．DB の名前や読み込むテキストファイルについてはプログラムを直接書き換えることで実行した．その結果作成された DB の中身を出力した結果を以下に示す．

```
C:\Users\Owner>sqlite3 rugby.db
SQLite version 3.28.0 2019-04-16 19:49:53
Enter ".help" for usage hints.
sqlite> .table
texts
sqlite> select * from texts;
1|Rugby is a sport
2|RugbyWorldCup is held in Japan
3|Japan is ranked 8th in the world
4|Japan won against Russia
5|Japan won against Ireland
```

表 6: 作成したデータセット (rugby.txt)

Rugby is a sport
RugbyWorldCup is held in Japan
Japan is ranked 8th in the world
Japan won against Russia
...
England lose to SouthAfrica
...
JamieJoseph is the coach of Japan
YuuTamura scored 51 points
...
MichaelLeitch tackled 44 times
England is ranked 1st in the world
NewZealand is ranked 2nd in the world
Wales is ranked 3rd in the world

```
6|Japan won against Samoa
7|Japan won against Scotland
8|England lose to SouthAfrica
9|Japan lose to SouthAfrica
10|JamieJoseph is the coach of Japan
11|YuuTamura scored 51 points
12|KotaroMatsushima scored 25 points
13|KenkiFukuoka scored 20 points
14|PieterLabuschagne tackled 68 times
15|JamesMoore tackled 67 times
16|ShotaHorie tackled 58 times
17|KazukiHimeno tackled 50 times
18|KeitaInagaki tackled 48 times
19|LukeThompson tackled 47 times
20|MichaelLeitch tackled 44 times
21|England is ranked 1st in the world
22|NewZealand is ranked 2nd in the world
23|Wales is ranked 3rd in the world
sqlite>
```

これは作成したテキストファイルと同様の内容であることが確認できた．

次に，この DB を用いて検索を行った結果を以下に示す．今回のデータセットでは，Unify クラス，Matching クラスともに main メソッドを作成し，そこからそれぞれでプログラムを呼び出しテキストファイルから DB への値の格納・マッチングを行う．これらのプログラムの連動関係に関しては増田君がアーキテクチャを考えたため，そちらのレポートを参考にされたい．

以下に Matching クラスと Unify クラスの main メソッドを用いて作成したデータセット読み出して動作させた場合の検索実行例を示す．Matching クラスでは検索文 (パターン) は 1 つで，Unify クラスは 2 つで検索を行う．しかし同一メソッド searchData を用いるため，検索文が 2 つの場合は「,」で検索文を区切り，実行時に分割して検索を行うこととした．

```
1  C:··· >java -cp sqlite-jdbc-3.21.0.jar;. Matching "Rugby
       is a ?x"
2  Successfully started
3  検索結果を取得
4  ★answer = sport
5  C:··· >java -cp sqlite-jdbc-3.21.0.jar;. Matching "Japan
       is ranked ?y in the world"
6  Successfully started
7  検索結果を取得
8  ★answer = 8th
9  C:··· >java -cp sqlite-jdbc-3.21.0.jar;. Matching "Japan
       won against ?z"
10 Successfully started
11 検索結果を取得
12 ★answer = Russia
13 ★answer = Ireland
14 ★answer = Samoa
15 ★answer = Scotland
16 C:··· >java -cp sqlite-jdbc-3.21.0.jar;. Unify "?x won
       against Russia,?x lose to ?y"
17 Successfully started
18 検索結果を取得
19 ★answer = ?x0 = Japan
```

```
20 ★answer = ?y0= SouthAfrica
```

作成したデータセットと比較しても，正しい結果が得られていることが確認できたので，DB から受け取った値と検索文との比較検索が正しく行われ，正確な結果が得られていることが分かった．

5.3 考察

データセットの登録方法はどのような方法でも良いと記述されていたため，テキストファイルに直接書き込む形で作成を行った．記述に際しては，今回は複雑になりすぎないよう Matching クラスを利用してマッチングを行うことを考え，似たような語彙や記述が入るように工夫した．また，国名や人名はスペースを空けると別の語として扱われてしまうため，これを防ぐためスペースを空けずに記述することとした．しかし，今回のデータセットでは中々解 (answer) が複数になる結果が得られるようなデータセットにならなかったため，今後データセットを作成する際には解が複数になるような可能性も多く考えられるようなデータセットを選択したい．また今回のデータセットでも，選手のポジション等を利用して作成すればよりよいデータセットにすることが出来たのではないかと実行を行って感じた．

二つのパターンによる検索も行えるデータセットになるよう作成したが，検索可能なパターンの数が減ってしまったため，この検索の種類ももう少し増やして作成できると実行結果が確認しやすくなったのではないかと考えた．

形式は必須課題 2-1 と同じであるため，DB への格納等はテキストファイルが変わっても正しく動作することも確認できた．また今回は読み込むテキストファイルや DB を変更する場合は直接プログラムを書き換える必要がある．これは入力の操作でこれらを変更できたりするようプログラムを書き加えることで，プログラムを直接触ることなく読み取りファイルの変更が可能になると考えたが，実装する時間を取ることが出来なかった．しかしプログラム方法自体は比較的容易だと考えられるので，時間に余裕があれば取り組んでみたいと考えた．

6　課題 2-3

上記システムの GUI を作成せよ.
・データの追加，検索，削除を GUI で操作できるようにすること.
・登録されたデータが次回起動時に消えないよう, 登録されたデータをファイルへ書き込んだり読み込んだりできるようにすること.

私の担当は全体のシステム設計と Presenter の構築に留まるため, 実行例は無い.
実装と考察に関しては適宜, 各種手法の詳細説明において言及する.
また, 必要に応じてソースコードを明示する場合もある.

6.1　手法

今回の課題では, データベースの利用や GUI の設計などシステムを構成する要素が前回課題よりも多いと考えられる. したがって, 機能区分を明確に行うことによって, システムとしての保守性やソースコードの可読性, 依存関係の明確化を狙うことが重要であると考えた. これらの目的を達成するために, 以下のような一般的にシステム・アプリケーション開発で取り入れられている設計概念を採用することとした.

MVP アーキテクチャ

DAO パターン

抽象化による疎結合な関係性の構築

6.2　MVP アーキテクチャの導入

課題 2-3 では,GUI の使用やデータベースの応用が必要となるため, 明確な機能区分を与えることで全体の設計指針が立つと考えられる. そこで, 今回はアプリケーションアーキテクチャとして代表的な MVP アーキテクチャを採用することによって, 機能の切り分けを行った. 以下に MVP アーキテクチャの各構成要素について述べる.

Model データ処理機構を担っている. 今回はデータを格納するデータベースやデータベースへのアクセスを担う DAO が該当する.View や Presenter に依存しない.

Presenter View から受けた処理に基づいて Model からデータを取得し, 画面反映を行うための View メソッドを呼び出す. これにより,View と Model 間の円滑なデータフローと画面制御を行う.

View ユーザーインターフェースを担う. ユーザー入力を受け取り,Presenter に通知し, 処理結果を出力として画面に表示する. 今回は, 入出力に対応するロジックで構成される GUI が該当する.

以下に MVP アーキテクチャの概念図を示す.

図 2: MVP アーキテクチャの概念図

6.3 DAO パターンの導入

今回は, データベースへのアクセスを一括して担うデザインパターンである DAO を導入した. 全てのデータ処理において, 必ず DAO を通すことによって, 他のクラスにデータベースアクセスメソッドが分散することを防ぐことができる. さらに,Model としての機能を担っていると考えられるため,Presenter のみが DAO インスタンスを握るように設計した. ただし, テキストファイルの使用が課題の条件として提示されていたため, 通

常アプリケーション開発では行われないテキストファイル-データベース間の処理機構も必要となった. これに対処するために, 今回は通常の DAO に加え, テキストファイル-データベース間の処理を担当する TextDAO を設計した. これら二種類の DAO の分類は, 以下の考えに基づいて明確に規定される.

DAO 通常のデータベースアクセス処理を担い, 今回はデータの追加・取得・削除を行うメソッドを取りまとめている.

TextDAO データベースをテキストファイルの一次キャッシュとみなし, テキストファイル-データベース間の読み書きを担当する. 具体的には,GUI 起動時にテキストファイルからデータベースへのデータを読み込み,GUI 終了時にデータベースからテキストファイルへのデータ書き込みを行う. 結果として, テキストファイルとデータベースの一貫性を保つこととなる.

以下に DAO パターンの概念図を示す.

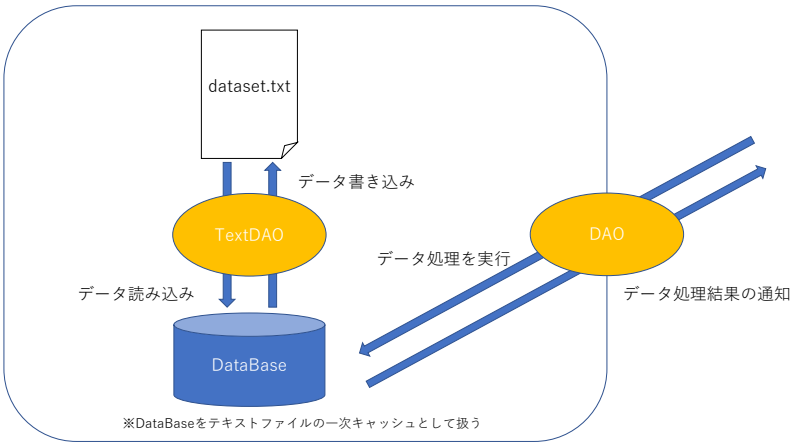

図 3: DAO パターンの概念図

今回,Presenter は TextDAO に対して以下のメソッドを要求する.

readTextFile メソッド テキストファイルからデータベースへのデータ読み込み

writeTextFile メソッド データベースからテキストファイルへのデータ書き込み

Presenter は DAO に対して以下のメソッドを要求する.

addData メソッド データベースにデータを追加する

fetchDataList メソッド データベースからデータを取得する

deleteData メソッド データベースからデータを削除する

6.4 抽象化による疎結合な関係の構築

まず,MVP アーキテクチャにおける各要素の間の依存関係について述べる. 上述した通り,Model クラスは Presenter や View に全く依存していない. 一方で,Presenter が Model インスタンスを握っていることから,Presenter は Model に依存する関係にある. 加えて,MVP アーキテクチャ含め,一般的な MVC 系統のアーキテクチャにおいては,View が Controller 部のインスタンスを握ることから, 今回は View が Presenter インスタンスを握ることとなる.
ここで,MVP アーキテクチャの特徴として,Presenter はユーザー入力を Model に通知するだけで無く, 直接 View への反映命令を行う点があげられることに注目する. 後者の機能の実現には, 何らかの形で Psesenter から View メソッドを呼び出す必要性が生じる. したがって,Presenter 内において,View メソッドを持ちうるインスタンスの生成が不可避的となる. しかし,View と Presenter が直接互いのインスタンスを握り合うことは望ましく無い. 何故ならば, これらの要素が互いのインスタンスをにぎり合うことにより, 強結合と呼ばれる相互的な依存関係が生まれ, アーキテクチャを採用することによる機能分割の意味が薄れるためである. 実際, 互いに直接インスタンスを握り合うのであれば, アーキテクチャという観点からはクラスを統一した場合と意義的にはほぼ等しい.
この問題を解決するための手法が, インターフェースを利用した抽象化による疎結合な関係の構築である. 実装としては,Presenter から View を管理するメソッドを集めた ViewInterface を定義し,Presenter 側では ViewInterface 型インスタンスを持つようにする.

ソースコード 15: ViewInterface の定義

```
    interface ViewInterface {
        \\データベース初期化完了メソッド
        void successStart();
        \\テキストファイル記録完了メソッド
        void successFinish();
        \\データ追加完了メソッド
        void successAddData();
        \\検索結果反映メソッド
        void showSearchResult(List<TextModel> resultList);
        \\データ削除メソッド
        void successDeleteData();
        \\一覧表示メソッド
        void showResultList(List<TextModel> resultList);
        \\例外処理表示メソッド
        void showError(String errorText);
        \\データ無し表示メソッド
        void showNoData();
    }
```

ソースコード 16: Presenter における ViewInterface 型インスタンスの生成

```
    class Presenter {
        ...
        private ViewInterface view;

        public Presenter(ViewInterface view) {
            this.view = view;
        }
        ...
    }
```

View 側の実装クラスでは, インターフェースを実装し, オーバーライドされたメソッドの具体的な処理を記述する.

ソースコード 17: GUI における Presenter インスタンス生成

```
    class GUI implement ViewInterface {
    ...
    Presenter presenter;
    init() {
        presenter = new Presenter(this);
    }
    ...
}
```

また,View は生成後に Presenter コンストラクタに自身を渡すことで,Presenter インスタンスを生成する．反対に View の終了時には,Presenter インスタンスを離すように設計する．

以上により,View が Presenter インスタンスを握ることで依存しながらも,Presenter 側は直接明示的に View インスタンスを握らずとも View メソッドを呼び出せる状態が完成する．すなわち,View が Presenter を持つという一方的な依存関係としての疎結合な関係の構築が達成される．
結果的に, 一般的にアーキテクチャを採用する際に好ましいとされる緩やかな関係性を持たせることができるのである．
以下に,View-Presenter 間における疎結合な関係を示す．

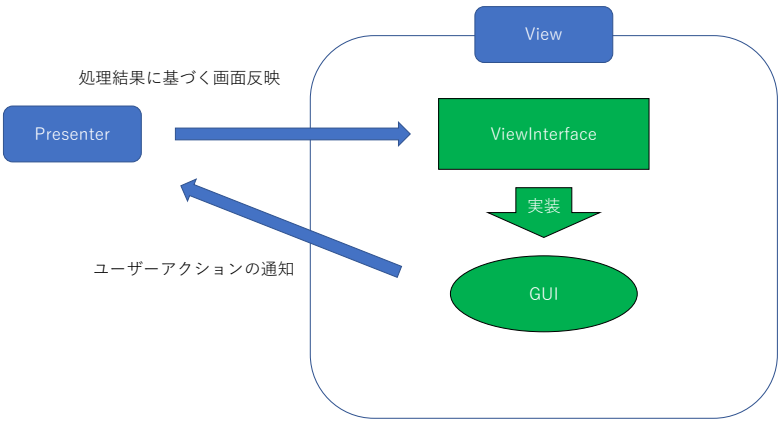

図 4: View 内部と Presenter の依存関係の概念図

6.5　システムの全体像

上述した 3 種類の技術を用いて, 今回の課題のシステムを実現した. その全体像を以下に示す.

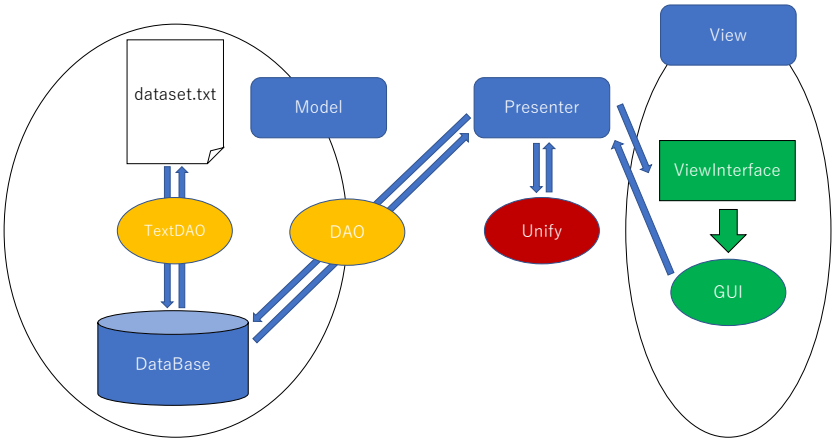

図 5: システム全体の概念図

6.6　データ追加・削除機能の実装

View がユーザー入力を受け取り,Presenter に通知する. Presenter は DAO に問い合わせ, 結果により呼び出す View メソッドを切り替えて View に反映指示を出す.

6.7　データ検索・一覧表示機能の実装

View がユーザー入力を受け取り,Presenter に通知する. Presenter は DAO に問い合わせ, 結果として受け取ったデータリストとユーザー入力から得られた文字列を Unify 内部のメソッドに渡すことで検索を行う. Unify による検索結果から, 呼び出す View メソッドを切り替えて View に反映指示を出す.

6.8 テキストファイル-データベース間の読み込み/書き込み機能の実装

View がユーザー入力を受け取り,Presenter に通知する． Presenter は TextDAO に問い合わせ, 実行結果により呼び出す View メソッドを切り替えて View に反映指示を出す.

6.9 考察

ここでは, 全体的な考察について述べることとする.
まず, アプリケーション/システムとして重要なのは機能を明確に区分することであると考えた. 次に, チーム開発を行うに当たって, 保守性や可読性を高めることに重点を置くべきだと考えた．これら二つの考えを基として,MVP アーキテクチャや DAO パターンの導入が適切であると判断した．その反面, 概念的な要素が多く, チームメンバーに対しての説明を要した．しかし,GitHub 上でソースコード管理を行っているので, コンフリクションを防止するためにもアーキテクチャやデザインパターンにより役割分担を行うことは非常に有効な手段であると考えられる.

さて, 機能の切り分けと分担を考える上では非常に良い課題であると感じた反面, 今回の課題について, 非常に違和感を感じた点がある．それは, 課題 2-3 において, テキストファイルへのデータ読み込みと書き出しによりデータの保管を行う指示があることだ. 私は, 今まで Android アプリケーションの開発に携わってきたが, まずデータ保管のためにテキストファイルを用いるのは一般的では無いとの認識を持っている. 通常, サーバーに管理を任せるか,OS 標準搭載のデータベースを用いる．今回であれば, 課題 2-1 でデータベースの作成を要求されているので, データベースを利用するのみで十分であると考えられる. また, データベースの永続性の観点から, わざわざテキストファイルへの保存を行うのは冗長であると感じられる. 課題の指示内容の解釈に苦しんだので, グループメンバーが TA に質問して提案された内容を重んじ, データベースをテキストファイルの一次キャッシュのような形にすることを決定した. 上述した通り, データベースの利用法としては非常にナンセンスだと感じられる.
次に,Unify とデータベースの検索機能が競合する点である. Unify は読み込んだデータを元に指定された文字列とのマッチングを行うことで検索

を行う．一方で, データベースに動詞ごとのテーブルを作り, 主格と目的格を属性として持たせ,SELECT 文を実行することで同様の機能を実装できる．すなわち, 検索機能に関して二つの実装方式が可能であり, 問題文からは Unify を用いる実装, 一般的にはデータベースを用いる実装があげられる．前者を採用した場合は, データベースはただ一つのみのテーブルを持ち, そのまま文章を格納するだけの機能となり, 十分にデータベースを活用できているとは言えない．後者を採用した場合は, そもそも Unify を利用する必要性が無くなり, 課題の趣旨に反する可能性がある．結果的に, 今回の課題テーマにあるパターンマッチングを活かすために前者を採用したが, 現実的な開発においては後者が最適であると個人的に強く感じる．
これらのような課題指示の曖昧性と現実的なシステムからの乖離は学生を非常に混乱させ, メンバー間の意思疎通をいたずらに困難とすると考えられる．もちろん, 実装課題であるので, 受け手側の受け取り方による多少の揺らぎは許容されるべきであるが, 学生の知識や経験にそぐわない実装を手段として選ばざるを得ないような表現にはいささか疑問が残る．課題内容の吟味と説明の徹底を行うことを強く推奨すると共に, このように学生が解釈に手間取る可能性を十分に考慮して, 課題内容を早期に公開すべきであると考える．

7　発展課題 2-3

上記システムの GUI を作成せよ．
・データの追加，検索，削除を GUI で操作できるようにすること．
・登録されたデータが次回起動時に消えないよう, 登録されたデータをファイルへ 書き込んだり読み込んだりできるようにすること．

私は登録されたデータが次回起動時に消えないよう，登録されたデータをファイルへ書き込むことが出来るようにするためのメソッドの作成と，GUI からの DB アクセスのために作成した Dao.java の実装を行った．

7.1　手法

必須課題 2-1 で，テキストファイルからデータセットを読み込むときに利用した TextCon.javat と TextDAO.java を利用して新しい機能を追加し

た．追加したものは以下のようになっている．

1. GUI からの命令で書き換えが行われた DB から，データを取得する．
2. DB から取得したデータを，テキストファイルに書き込む．

これらの実装を私が担当した．

7.2 実装

書き換え終了後に DB からその内容を取得するメソッド getDBData，取得したデータをテキストファイルに書き戻すメソッド writeTextFile の作成を行った．それぞれのメソッドは，getDBData メソッドが TextDAO.java に，writeTextFile メソッドが TextCon.java に含まれている．

取得したデータをテキストファイルに書き戻すメソッド writeTextFile の実装をソースコード 18 に示す．

ソースコード 18: writeTextFile メソッド

```
1  public void writeTextFile() throws FileNotFoundException{
2          try { // ファイル読み込みの操作
3                  ...
4                  // データの取得【DB】
5                  TextData = TextDAO.getDBData();
6                  TextDAO.conCom();
7                  // ファイルへの書き込み
8                  for (int i = 0 ; i < TextData.size() ; i
                       ++){
9                          out.println(TextData.get(i));
10                 }
11                 out.close(); // ファイルを閉じる
12         } catch (IOException e) {
13                 e.printStackTrace();
14         }
15         TextDAO.closeConn();
16 }
```

DB に格納されているデータを取得するメソッド getDBData の実装をソースコード 19 に示す．

ソースコード 19: getDBData メソッド

```
1  // DB のデータを取得
2  public static ArrayList<String> getDBData() {
3          conn = null;
4          ArrayList<String> DBList = new ArrayList<String
               >();
5          PreparedStatement pStmt = null;
6          ResultSet rs = null;
7          String sql;
8          try{
9                  Class.forName("org.sqlite.JDBC");
10                 if(conn == null){
11                         conn = DriverManager.getConnection(
                               connDB);
12                         conn.setAutoCommit(false);
13                 }
14                 sql = "select line from texts";
15                 pStmt = conn.prepareStatement(sql);
16                 rs = pStmt.executeQuery();
17                 while (rs.next()) {
18                         String text = rs.getString("line
                               ");
19                         DBList.add(text);
20                 }
21                 rs.close();
22                 pStmt.close();
23         }catch{
24         …
25         }
26         return DBList;
27 }
```

これらに加え，増田君が作成してくれたた DB へのアクセスを行う Dao.java のプログラムに対して実際の動作が正しく行われるように修正をしたり，Presenter.java の DB アクセス指令の改良やデバッグなども行った．

7.3 実行例

GUI の作成は青山君が作成したため，ここでは私が実装したファイルの書き込み部分のみに着目して実行例を確認する．必須課題 2-1 で作成し

た DB 作成テスト時に使用した Test.java を用い，テキストファイルを読み込んで DB に格納し，値の更新や削除を行った後変更した DB からテキストファイルへと上書きまでの動作が正しく行われたかを確認する．使用するデータセットは，与えられた dataset_example.txt とした．

dataset_example.txt を操作するための Text.java をソースコード 20 に示す．

ソースコード 20: Text.java の main メソッド

```
1  public static void main(String arg[]){
2          // DB の作成
3          Presenter presenter = new Presenter(new View());
4          presenter.start();
5          // DB の書き換え
6          presenter.addData("Mike is a boy");
7          presenter.deleteData(1);
8          presenter.addData("Mike loves Hanako");
9          presenter.deleteData(5);
10         presenter.addData("Mike has a hobby of playing
               tennis");
11         presenter.deleteData(20);
12         presenter.addData("Mike is a boy");
13         // テキストファイルへの上書き
14         presenter.finish();
15 }
```

addData でデータの追加を行い，deleteData でデータの削除を行う．これらはそれぞれ引数に String と int を持つ．また start() でテキストファイルから DB への書き込みを，finish() で DB からテキストファイルへの上書きを行う．これらは Presenter を経由してから DAO へと命令が伝えられる．これを実行するが，このとき変更された内容は繰り返し利用され，2 回目の実行時には 1 回目の変更内容が反映された状態で新しくプログラムの更新を行わなければならない．それを踏まえ実装を行った実行結果が以下のようになる．

上の Test.java の内容を 3 回繰り返して得られた結果を以下に順に示す．この時示すのは，結果が反映されたテキストファイルとする．

ソースコード 21: 実行前の dataset_example.txt

```
Hanako is a girl
Hanako is a student
student is a kind of human
human is a kind of mammal
Hanako has a hobby of playing video-games
Hanako has a hobby of playing air-guitar
Hanako studies philosophy
Hanako loves Taro

Taro is a boy
Taro is a student
Taro has a hobby of playing video-games
Taro studies informatics
Taro loves Jiro
Taro has a pet named Jiro

Jiro is a boy
Jiro is a dog
dog is a kind of mammal
Jiro has a hobby of playing frisbee
Jiro loves Hanako
```

ソースコード 22: 1 回実行後の dataset_example.txt

```
Hanako is a student
student is a kind of human
human is a kind of mammal
Hanako has a hobby of playing air-guitar
Hanako studies philosophy
Hanako loves Taro
Taro is a boy
Taro is a student
Taro has a hobby of playing video-games
Taro studies informatics
Taro loves Jiro
Taro has a pet named Jiro
Jiro is a boy
Jiro is a dog
dog is a kind of mammal
Jiro has a hobby of playing frisbee
Jiro loves Hanako
Mike loves Hanako
Mike has a hobby of playing tennis
```

```
20 Mike is a boy
```

ソースコード 23: 2 回実行後の dataset_example.txt

```
1  student is a kind of human
2  human is a kind of mammal
3  Hanako has a hobby of playing air-guitar
4  Hanako loves Taro
5  Taro is a boy
6  Taro is a student
7  Taro has a hobby of playing video-games
8  Taro studies informatics
9  Taro loves Jiro
10 Taro has a pet named Jiro
11 Jiro is a boy
12 Jiro is a dog
13 dog is a kind of mammal
14 Jiro has a hobby of playing frisbee
15 Jiro loves Hanako
16 Mike loves Hanako
17 Mike has a hobby of playing tennis
18 Mike is a boy
19 Mike loves Hanako
20 Mike has a hobby of playing tennis
21 Mike is a boy
```

ソースコード 24: 3 回実行後の dataset_example.txt

```
1  human is a kind of mammal
2  Hanako has a hobby of playing air-guitar
3  Hanako loves Taro
4  Taro is a student
5  Taro has a hobby of playing video-games
6  Taro studies informatics
7  Taro loves Jiro
8  Taro has a pet named Jiro
9  Jiro is a boy
10 Jiro is a dog
11 dog is a kind of mammal
12 Jiro has a hobby of playing frisbee
13 Jiro loves Hanako
14 Mike loves Hanako
15 Mike has a hobby of playing tennis
```

```
16 Mike is a boy
17 Mike loves Hanako
18 Mike is a boy
19 Mike is a boy
20 Mike loves Hanako
21 Mike has a hobby of playing tennis
22 Mike is a boy
```

7.4 考察

Test.java では，4 つの文の追加と 3 つの文の削除を行っている．このことから実行結果を確認すると，実行回数が増えるたびにテキストファイルに入っている文章の数が一つずつ増えていることが確認できる．また追加では Mike に関する情報を加えているが，実行回数が増えるたびに Mike に関する文章の数が増えていることも確認できる．これらのことから，このプログラムは前回実行時の内容を保持したまま正しく次の実行が行えていることが分かった．

DB 格納時に空行は飛ばして格納するようにしたことから，DB の内容をテキストファイルに格納する場合も空行は残らずに上書きが行われることも分かった．さらに，データの削除は DB 格納時に各文章に与えられる id 番号によって行われるが，実行するたびにテキストファイルの一番上にある行が消えていることから，DB に格納が行われるたびに与えられる id 番号が正しく与え直され，その時のテキストファイルの一文目が削除されてることが確認できた．

発展課題ではあったものの，私が担当した範囲は必須課題 2-1 で作成したプログラムと DB に対しての操作を行うため，実装方法自体は大きく悩むことなく取り組むことが出来た．また，GUI との連動に必要なメソッドは Presenter.java によって操作が行われるため，私自身は GUI 本体を操作することなくプログラムを動作させることが出来た．これはプログラム作成時のアーキテクチャを明確にしていたことが影響しているため，プログラムを作成する前にアーキテクチャを明確にすることの大切さを強く理解した課題となった．

8　DB関連についての全体の考察

ここまで，各課題に対する考察を行ってきたため，ここからは私が担当したDB全体に対する考察を行う．

今回の課題はDBを作成するとの課題であったことから，プログラミング応用の講義で利用したsqliteを使用してDBを作成することを考えた．課題の要件を満たすようなDBの形として，最終的に表1の格納方法を使用した．この理由としては，実装難度と実装期間と発展課題でデータの追加や削除を行う必要があると考えたため，DBを利用するとこれらの実装が行えると考えたからである．しかしこれでは，動作が終了するとDBで読み込んだ内容を最終的にテキストファイルへと保存することになり，DBはデータを保存するという意味では機能しなくなってしまう．DBは本来テキストファイルよりもデータの保存に適したものであるのにも関わらず，DBをテキストファイルのキャッシュのように使用することになってしまい，これではDBを利用する利点が無くなってしまうと考えられた．

さらに，DBはtableやカラムを持つことが出来ることもテキストファイルよりも保存方法に幅がある．しかしこの格納方法ではほとんどテキストファイルと同様の保存方法になっており，これもDBを有効に活用できていないと感じられる原因となった．

これらを解決するためにも，単語毎で分割し動詞毎でtableを作成したり，単語毎でフィールドに追加していく方法も考えた(必須課題2-1考察参照)．しかしこの方法を利用すると，検索時にMatchingクラスやUnifyクラスを利用せずとも，変数が単語の分割された何個目に含まれているかという情報と各単語の内容で比較を行うことで検索が出来てしまうと考えられる．これでは課題の内容を満たしているとは言えないため，この方法は使用しないことを決めた．

MatchingクラスやUnifyクラスの利用の観点から考えると，先に検索を行い，その際の検索文とその答えをDBに格納する方法が課題内容も満たしDBも活用できているため最も良いようにも考えられる．しかしこれでは発展課題のデータの追加や削除がDBで行えないため容易には実装出来なくなってしまう．追加や削除が行われたデータセットを最終的に保存しなければならないことを考えると，この方法も課題内容を満

たさないと考えた．

このような考えから，今回は表 1 のように実装を行ったが，やはり DB を活用できているようにはあまり感じられない結果となった．そのため，時間に余裕があれば他の方法での実装も検討してみたいと考えた．例えば，検索文とその答えの格納を DB に行う場合は，初めにテキストファイルを読み込んだ際にその内容を List などに格納しておき，検索や追加，削除はその List を利用して行うようにし，最終的には変更された List の内容をテキストファイルに上書きする形を取ることで，発展課題の内容も満たすことが出来るのではないかと考えた．

また，データの追加や削除と課題にあったが，この「データ」とはデータセットを指すのか，検索文 (パターン) を指すのかも明確に指定されていないように感じた．今回私たちの班は「データ」をデータセットと考えたが，パターンと考えて実装を行うことも可能であると考えられた．そのため，検索文 (パターン) とその答えを格納する DB の作成は，課題の内容をより満たしたプログラムの作成となるのではないかとも考えた．

ここまでは DB の定義の仕方について考察したが，ここからは実際に作成した DB の実装について考察する．

DB の作成でまず工夫した点は，id 番号を引数で指定することなく決定することが出来るようにしたことである．テキストファイルから読み込んで一文ずつ DB に格納する際には 1 で初期化された変数を順にインクリメントすることで実現できる．しかし追加では別の DAO から DB にアクセスするため変数を取得することが難しい．そこで table のなかで最大の id を取得する関数 max(id) を利用するメソッド getNo を作成することで今使用されている最大の id 番号を取得し，それを 1 増やした値を追加されたデータの id 番号とした．

また，作成された DB の扱い方についても工夫を行った．初めに作成したプログラムでは，同じ table 名を作成しようとするとエラーが発生するため，一度実行して作成した DB は次の実行の前に自分で削除しなければならなかった．これを解決するため，まず作成した DB を実行終了と同時に消すことを考えた．しかし sqlite では DB そのものを消す命令が存在していなかったため，この方法は実現させることが出来なかった．

次に，実行終了時に DB ではなく table を消すことを考えた．これは drop table という命令で可能であることが分かったので，これを実装したところ，二回目以降の実行では table が消えているためエラーが発生しなくなった．しかしこれでは，1 回目の実行時に別の main で実行して残っていたプログラムが存在しているとエラーが発生することが分かった．また，実行終了時に table を消してしまうため，実行終了時に DB にはデータが残らなくなってしまう．そのため，この方法よりも良い実装を考えた．

その結果考えたものが，実行開始時に table を消去する方法である．しかしこれは table が存在しなかった時にエラーが発生してしまう．そのため table の存在の有無を判定しようと調べたところ，table が存在していないときにだけ table を新しく作成することが出来ることが分かった．ここから，table を作成した後にその中身を削除することを考えた．そこで deleteData メソッドを作成し，table の中身を全て削除してからテキストファイルのデータを書き込むことが出来るようになった．よって DB の有無にかかわらず，正しく読み込んだテキストファイルの内容を格納することが出来るプログラムを作成することが出来た．

これは求めたい結果が得られるまでにかなり遠回りをしてしまったが，プログラム改良のプロセスを体感できたように感じた．

今回作成した DB では，追加の際に既に入っているデータをもう一度追加すると，同一内容でもはじくことなく新しいデータとして格納してしまう．これではデータセットの内容の重複が認められてしまうことになるため，同一内容の文章ははじくための処理を入れるべきだった．Matching クラスや Unify クラスを用いれば内容が一致しているかどうかの判断は可能だと感じた．今回は時間がなくできなかったが，追加処理を行う際は現在入っているデータと追加文が重複しているかを確認し，重複していないときにのみデータへの追加を行うように実装したい．

また，Moodle の質問掲示板の返信から，そもそも既存の DB を利用せずに課題を解かなければならなかったことが分かった．そのため，今までの考え方を大きく変え自作 DB の実装方法を考えなければならなかったが，今回は時間がなく考えることが出来なかった．自作 DB の実装方法はまだどのように行えばよいのか明確に分かっていないため，今後のためにもどのような方法が利用できるのか考えていきたい．

9　発展課題 2-3

上記システム (Matching クラスまたは Unify クラスを用いた，パターンで検索可能な簡単なデータベース) の GUI を作成せよ．
データの追加，検索，削除を GUI で操作できるようにすること．
登録されたデータが次回起動時に消えないよう，登録されたデータをファイルへ書き込んだり読み込んだりできるようにすること．

私の担当箇所は，発展課題 2-3 の GUI 全般の Swing を用いた実装である．

9.1　手法

GUI を実装するにあたり，以下のような方針を立てた．

1. データベースとデータのやりとりをするためのクラスやメソッドを作る．
2. 検索・追加・削除のためのテキストフィールドやボタン，リストを表示する．
3. 表示した各種コンポーネントを動作させる．データベースからデータを受け取って GUI に反映する．

1．に関して，班員と協力してタスクを分割し，データベースとの直接のやり取りは Presenter クラスに任せた．自分は Presenter からデータを受け取るための View クラス等を作成して用いることで，より構造化されたデータのやり取りを可能とした．

2．に関して，GridBagLayout を用いてコンポーネントの配置を行うことで，ユーザがより直感的に利用できるよう工夫した．また，データベースの一覧を表示することで，データの追加・削除・検索の視覚的な確認を行えるような仕様とした．また，削除を一覧から選択して実行できるように，一覧の表示には JList クラスを用いた．

3．に関して，VIew クラスを介することで，コンソールを通じて GUI に正しく反映できているかを確認できるような仕様とした．また，ボタンを押したと同時に GUI 上のリストを更新するために，DefaultListModel クラスを利用した．

9.2 実装

実装にあたり，主に下記のサイトを参考にした．

TATSUO IKURA：『Swing を使ってみよう - Java GUI プログラミング』 https://www.javadrive.jp/tutorial/ (2019/10/29 アクセス)

GUI に大きく関連するプログラムとして，UnifyGUI.java, Presenter.java, TextModel.java, ViewInterface.java が挙げられる．各プログラムの説明については以下の通りである．
UnifyGUI.java には以下のクラスが含まれる．

- SearchGUI: メソッド main, actionPerformed, クラス myListener を実装したクラス．
- View: インターフェース ViewInterface を実装したメソッド，各種ゲッターを実装したクラス．

Presenter.java には以下のクラスが含まれる．

- Presenter: メソッド start, finish, addData, searchData, deleteData, fetchData を実装したクラス．

TextModel.java には以下のクラスが含まれる．

- TextModel: データベースの ID とテキストを一元的に保持するためのクラス．ID とテキストのゲッターを実装している．

ViewInterface.java には以下のクラスが含まれる．

- ViewInterface: メソッド successStart, successFinish, successAddData, showSearchResult, successDeleteData, showResultList, showError, showNoData を持つインターフェース．

9.2.1 データベースとデータのやりとりをするためのクラスやメソッドを作るまで

他の班員が作った Presenter クラスを介してデータを受け取るために，まず ViewInterface インターフェースを実装する必要があった．初めは

UnifyGUI 自体に実装しようと考えたが，Presenter のインスタンスの引数に VIewInterface を実装したクラスを渡す必要があったため，こうすると UnifyGUI の main メソッドを実行しようとしたとき，static であるため自身を引数として渡すことができなくなるという問題が発生した．

そこで，ViewInterface を実装するためのクラスとして View クラスを別に作った．View クラスでは Presenter とのやり取りのたびにコンソールに結果が出力されるため，デバッグ等で活用しやすいものにできた．

また，Presenter クラス側から，プログラム開始時に start メソッド，終了時に finish メソッドを呼び出してほしいという要求があったため，WindowAdapter クラスを拡張した myListener において，windowOpened メソッドと windowClosing メソッドの実装によりこれらを実現した．これにより，ウィンドウが最初に表示されたときに start メソッドが呼び出され，ウィンドウを閉じようとしたときに finish メソッドが自動的に呼び出されるように実装した．myListener クラスをソースコード 25 に示す．

ソースコード 25: myListener クラス

```
1   public class myListener extends WindowAdapter {
2       public void windowOpened(WindowEvent e) {
3           view = new View();
4           presenter = new Presenter(view);
5           presenter.start();
6           presenter.fetchData();
7           textList = view.getRl();
8           for (TextModel text : textList) {
9               lModel.addElement(text);
10          }
11      }
12
13      public void windowClosing(WindowEvent e) {
14          presenter.finish();
15      }
16  }
```

9.2.2 検索・追加・削除のためのテキストフィールドやボタン，リストを表示するまで

検索や追加には入力が必要なため，JTextField を用いて入力を可能とし，JButton クラスで対応するボタンの表示を行った．また，検索結果の

表示と，データベースの要素の表示には JList クラスを用いた．

これらのコンポーネントの表示には，ユーザが直感的に扱いやすくするためのレイアウトを考える必要があった．そこで今回の形式に最も適していそうな GridBagLayout クラスを用いることとした．また，細かな配置を行うために GridBagConstraints クラスを用いた．このクラスのフィールドである gridx や gridy でセルを設定したり，anchor でセル内の配置を調整することで，ユーザにわかりやすいレイアウトの構築ができた．これをソースコード 26 に示す．

ソースコード 26: UnifyGUI コンストラクタの一部

```
1         gbc.gridx = 0;
2         gbc.gridy = 0;
3         layout.setConstraints(text, gbc);
4
5         gbc.gridy = 1;
6         gbc.anchor = GridBagConstraints.NORTHEAST;
7         layout.setConstraints(btnPanel, gbc);
8
9         gbc.gridy = 2;
10        gbc.anchor = GridBagConstraints.CENTER;
11        layout.setConstraints(searchSp, gbc);
12        ...
13        gbc.gridx = 1;
14        gbc.gridy = 0;
15        gbc.gridheight = 3;
16        layout.setConstraints(listSp, gbc);
17
18        gbc.gridy = 3;
19        gbc.gridheight = 1;
20        gbc.anchor = GridBagConstraints.NORTHEAST;
21        layout.setConstraints(delButton, gbc);
```

9.2.3 表示した各種コンポーネントを動作させたり，データベースからデータを受け取って GUI に反映したりするまで

ボタンが押下されたときに入力データを引数として Presenter からメソッドを呼び出すことを，ActionListener インターフェースの actionPerformed メソッドの実装により実現した．このとき，どのボタンが押されたかを判別するために，getActionCommand メソッドを用いた．

削除においては，リストから選択して行えるように，getSelectedIndex メソッドを利用して，どの項目が選択されているかの取得を行い，Presenter への引数として渡した．

これらを行う actionPerformed メソッドをソースコード 27 に示す．

ソースコード 27: actionPerformed メソッド

```
1    public void actionPerformed(ActionEvent e) {
2        String cmd = e.getActionCommand();
3        String arg = text.getText();
4
5        if (cmd.equals("検索")) {
6            presenter.searchData(arg);
7            sModel.clear();
8            searchList = view.getSr();
9            for (String text : searchList) {
10               sModel.addElement(text);
11           }
12
13       } else if (cmd.equals("追加")) {
14           presenter.addData(arg);
15           lModel.clear();
16           presenter.fetchData();
17           textList = view.getRl();
18           for (TextModel text : textList) {
19               lModel.addElement(text);
20           }
21
22       } else if (cmd.equals("削除")) {
23           if (!listPanel.isSelectionEmpty()) {
24               int index = listPanel.getSelectedIndex();
25               TextModel val = (TextModel) listPanel.
                     getSelectedValue();
26               presenter.deleteData(val.getUUID());
27               lModel.remove(index);
28           } else {
29               System.out.println("削除失敗(未選択のため
                     ) ");
30           }
31       }
32   }
```

このソースコードに示されたように，リストへの表示は DefaultList-

Model クラスの add メソッドを用いて行った．また，ソースコード 25 の start メソッド内でも同様の処理が行われており，リストの初期状態，すなわちデータベースの初期状態がウィンドウ表示時に反映されている．

また，このソースコードから分かるように，データの要求は Presenter に対して行っている一方で，データの受け取りは View を介して行っている．これはデータベースの保守性を高めるための，Presenter クラス担当者の意向によるものである．

もしリストの中身が多くなり，画面内に全ての要素が一度で表示しきれないときにスクロールバーを表示するために，JScrollPane クラスを用いた．JList インスタンスをこのコンポーネント内に入れることで，期待通りの実装ができた．この際，コンポーネントを思った通りのサイズにするために，setPreferredSize メソッドで調整した．これをソースコード 28 に示す．

ソースコード 28: UnifyGUI コンストラクタの一部

```
1         sModel = new DefaultListModel();
2         searchPanel = new JList(sModel);
3         JScrollPane searchSp = new JScrollPane();
4         searchSp.getViewport().setView(searchPanel);
5         searchSp.setPreferredSize(new Dimension(200,
              300));
6         ...
7         mainPanel.add(searchSp);
```

9.3　実行例

UnifyGUI を実行したところ，下図のような画面が得られる．

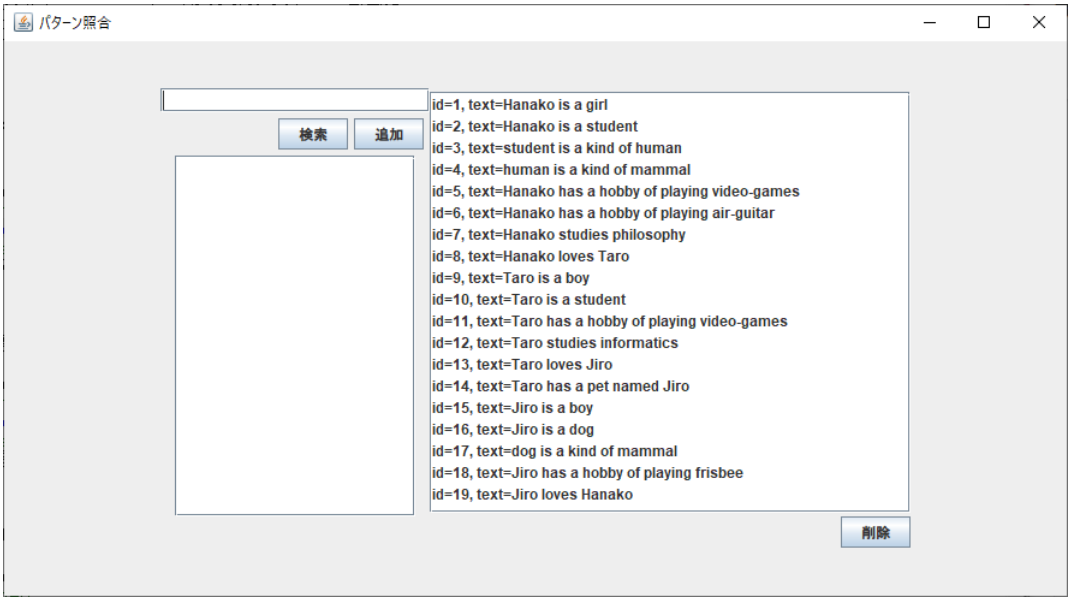

図 6: 初期状態

Matching に関する質問文「?x has a hobby of playing video-games」を入力し，検索ボタンを押したところ，下図のような画面が得られる．

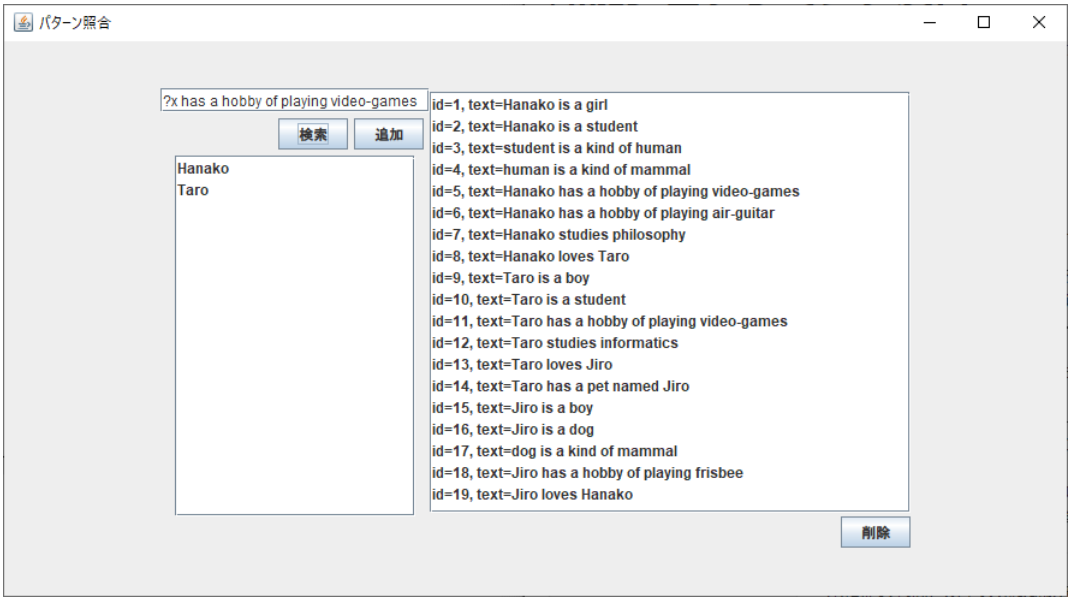

図 7: Matching の検索

Unify に関する質問文「?x is a boy,?x loves ?y」を入力し，検索ボタンを押したところ，下図のような画面が得られる．

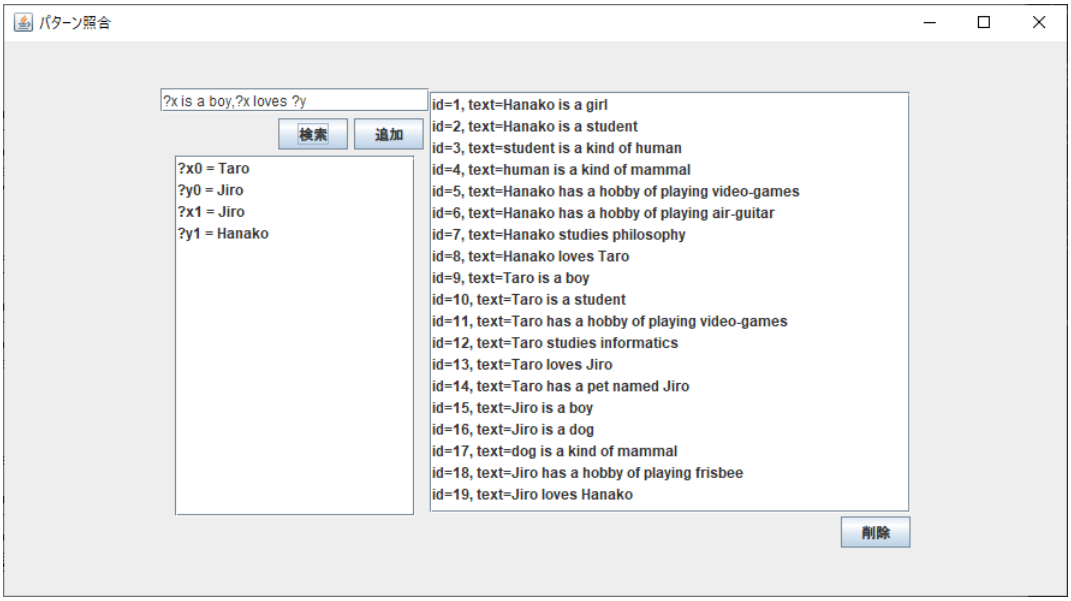

図 8: Unify の検索

データ「Shuhei is a boy」を入力し，追加ボタンを押したところ，下図のような画面が得られる．

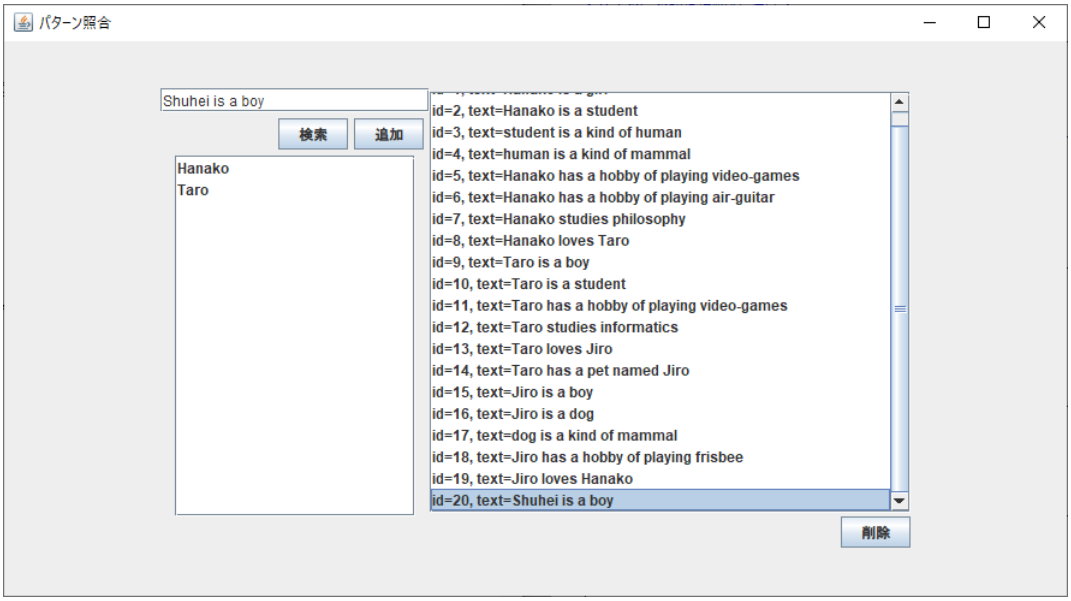

図 9: データの追加

右のリストから「id=19,text=Jiro loves Hanako」を選択し，削除ボタンを押したところ，下図のような画面が得られる．

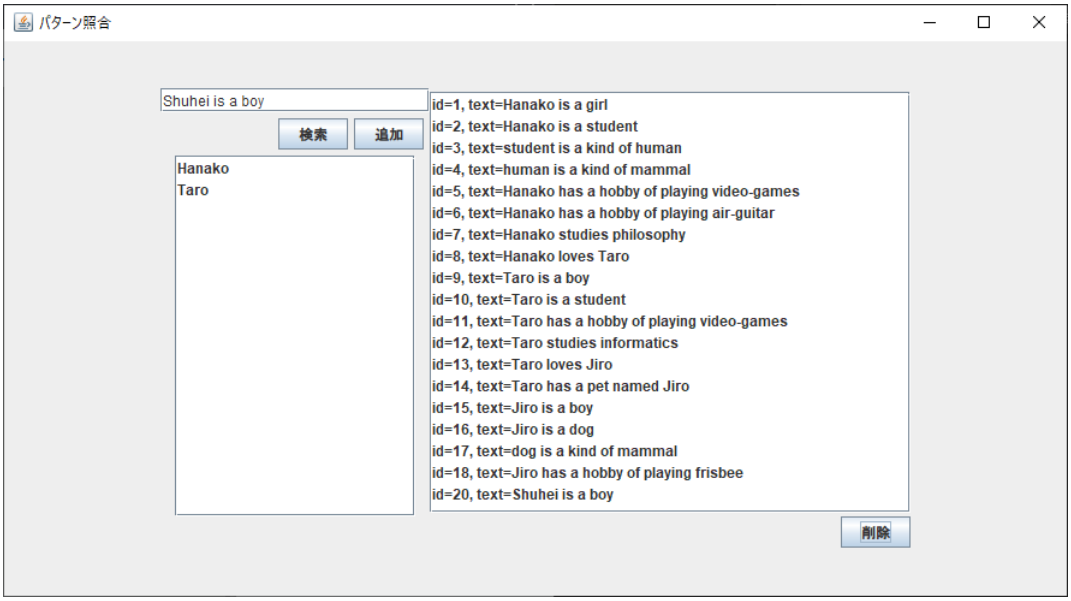

図 10: データの削除

右上の「×」ボタンからプログラムを終了し，再度 UnifyGUI を実行して起動したところ，下図のような画面が得られ，前回の追加・削除したデータが保持されていることが分かる．

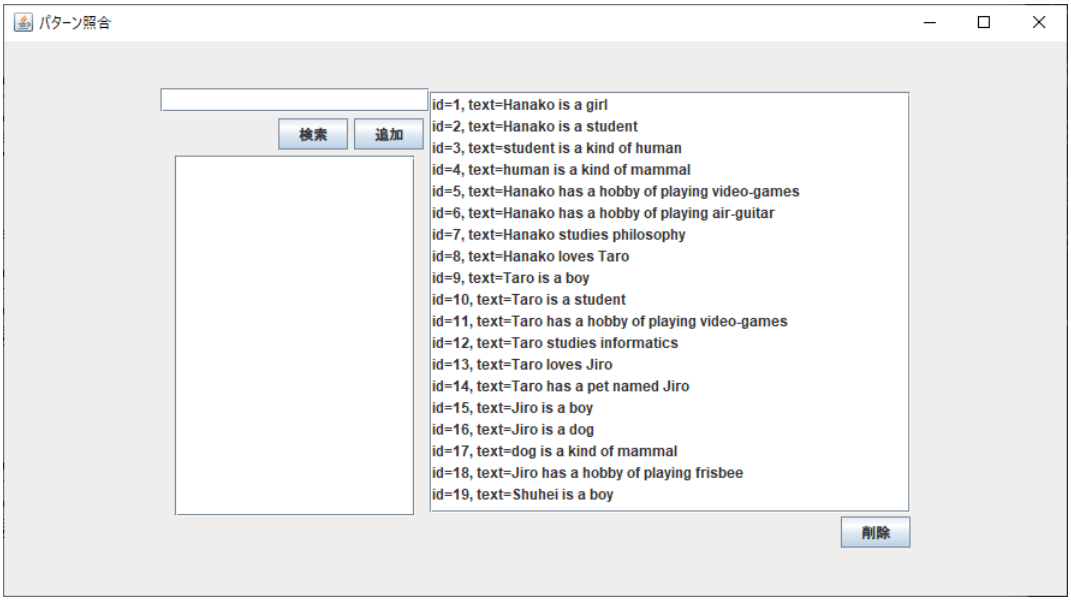

図 11: プログラムの再起動

9.4 考察

ViewInterface を実装するためのクラスとして View クラスを UnifyGUI クラスとは別に作ったが，これにより Presenter からの値を GUI 実行時に使えるようになっただけでなく，UnifyGUI クラスを介さずとも Unify クラス等の単体テスト時にも Presenter クラスを利用できるようになったため，結果として独立したクラスで ViewInterface を実装してよかったと考えられる．

また，Presenter 担当者からの要求として start,finish メソッドの実行をプログラム開始･終了時に求められたが，finish メソッドの呼び出しにおいて，windowClosing メソッドの他に，windowClosed メソッドの利用が候補として挙げられた．しかし，windowClosed メソッドでは finish メソッドの呼び出しが行われなかったため，windowClosing メソッドを用いることとなった．windowClosing メソッドはウィンドウをクローズしようとしたときに呼び出される機能である一方で，windowClosed メソッドはウィンドウがクローズされたときに呼び出される機能であるが，いずれにおいてもウィンドウはクローズされるのに，なぜ windowClosed メソッドでは finish メソッドが呼び出されないのかは，ウィンドウがクローズした後 finish メソッドを呼び出す前に UnifyGUI プログラムが終了してしまうためだと考えられる．要するに,「×」ボタンが押された時にプログラムを終了させるために設定した setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE) の方が，windowClosed メソッドよりも実行優先度が上であるためだと考えられる．また，コンソールから「Ctrl + C」でプログラムを強制終了した場合も，finish メソッドは呼び出されなかったため，強制終了は windowCloseing メソッドよりも優先度が高いことが分かった．

GUI の表示において，レイアウトの設定が大変であった．setPrefferdSize や setMaximumSize 等のメソッドを駆使しても，コンポーネントを意図した通りの配置や大きさにすることが難しかった．そこで，GridBagLayout クラスを用いたところ，これらの問題を一同に解消することができた．GridBagLayout は表形式でありながらも相対的な位置を高い自由度で配置することができるため，今後も自由な表形式の GUI を作りたいときに重宝すべきものであると考えられた．

JList にスクロールバーを追加するために JScrollPane クラスを用いたが，これによりペインにも様々な種類があることを認識できた．JScrollPane クラスを用いて実際に実装してみることで，ペインはパネルを多機能化したものというような認識を得ることができた．

今回のプログラムでは，データベース担当，GUI 担当，これらの仲介担当といったようにグループ内で分担を行ったが，この並行作業は大変なものであった．自分の知らないところでプログラムの構造が想定していたものから変わっていることがあり，グループ内で作業進捗の共有をより密に行う必要があったと感じたと同時に，プログラムが大規模になってくると，どうしても自分の干渉できない部分が出てくることは仕方のないことと割り切り，自分の想定していたものとは異なっていてもこちらが相手に合わせてあげるといったことも重要だと考えられる．

しかし，今回の他の班員が作ったデータベースの構成についてはもう少し言及すべきだったと反省している．今回のデータベースはキャッシュとして扱われており，これは永続的な記憶を目的とするデータベースの本来の用途とは異なっていると考えられる．また，データベースの構造自体も，各行に ID を割り振って 1 つのレコードとして取り扱うという，文が構造化されて分解されたものだとは到底言い難いものとなってしまったと考えられる．

参考文献

[1] Java による知能プログラミング入門 –著：新谷 虎松

[2] java CSV 出力 –著：TECH Pin
https://tech.pjin.jp/blog/2017/10/17/【java】csv 出力のサンプルコード

[3] Web アプリケーション開発者から見た、MVC と MVP、そして MVVM の違い (Qiita) –著：shinkuFencer
https://qiita.com/shinkuFencer/items/f2651073fb71416b6cd7 （2019 年 10 月 25 日アクセス）.

[4] 開発者が知っておくべき、6 つの UI アーキテクチャ・パターン ―― 「matarillo.com」より ―― (.NET 開発者中心 厳選ブログ記事) –著：猪股 健太郎
https://www.atmarkit.co.jp/fdotnet/chushin/greatblogentry_10/greatblogentry_10_01.html （2019 年

10 月 25 日アクセス）.

[5] お前らが Model と呼ぶアレをなんと呼ぶべきか。近辺の用語 (Entity とか VO とか DTO とか) について整理しつつ考える (Qiita) –著：takasek
https://qiita.com/takasek/items/70ab5a61756ee620aee6
（2019 年 10 月 25 日アクセス）.

[6] Java SQLite に JDBC 接続して insert/update/delete するサンプル (ITSakura)
https://itsakura.com/java-sqlite-insert （2019 年 10 月 25 日アクセス）.

[7] プログラミング応用で作成した DAO 等のプログラム

[8] RUGBY WORLD CUP, https://www.rugbyworldcup.com/（2019 年 10 月 29 日アクセス）.

[9] DBOnline SQLite 入門, https://www.dbonline.jp/sqlite/（2019 年 10 月 29 日アクセス）.

[10] memocarilog, https://memocarilog.info/php-mysql/5355（2019 年 10 月 29 日アクセス）.

[11] プログラミングマガジン　【デザインパターン】DAO/DTO パターン, www.code-magagine.com/?p=1311 （2019 年 10 月 29 日アクセス）.

[12] CMO クラウドアカデミー　データベースの用語を理解しよう「テーブル」「レコード」「カラム」「フィールド」とは?, https://academy.gmocloud.com/know/20160425/2259（2019 年 10 月 29 日アクセス）.