```java
package use_case_controller;

import java.awt.Color;

/**
 * Class NewCaseController to manage various validation and control flows of managing Caseload.
 * @author Daria Vekic (Student ID: 586661)
 *
 */
public class NewCaseController {

    //Error messages to be displayed - taken from Solution Planning report
    public String error2 = "Field cannot be blank.";
    private String error4 = "Case title must be in format "
                            + "[client surname] v [opponent] using "
                            + "alphabetical characters only.";
    private String error5 = "Name field cannot contain numbers.";
    private String error6 = "DOB invalid. DOB must be entered in format DD/MM/YYYY.";
    private String error7 = "DOB must contain valid numeric values.";
    private String error8 = "Postcode invalid.";
    private String error9 = "Phone number invalid. Must contain digits only "
                            + "and cannot contain whitespace or special characters.";
    public String error11 = "Invalid fields. Please review.";

    private CommonElements commonElements = new CommonElements(); //to access common elements

    //Instance field to hold number of invalid fields, if any.
    //Used to control if a case is added to List or not.
    private int invalidFields = 0;


    /**
     * Method to add a Case to List.
     * @param position the given Solicitor for this Case.
     * @param emps the List of Employees to be searched.
     * @param empName the name of the Solicitor to search for.
     * @param cases the List of Cases to be added to.
     * @param title the title of the new Case.
     * @param type the type of the new Case.
     * @param desc the description of the new Case.
     * @param clients the List of Clients to be added to.
     * @param fName the Client's first name.
     * @param lName the Client's last name.
     * @param dob the Client's date of birth.
     * @param address1 the Client's address first line.
     * @param address2 the Client's address second line.
     * @param postcode the Client's postcode.
     * @param phoneNum the Client's phone number.
     * @throws ParseException thrown if the DOB field cannot be parsed.
     */
    private void addCase(int position, ArrayList<Employee> emps, String empName,
            ArrayList<Case> cases, String title, CaseType type, String desc,
            ArrayList<Client> clients, String fName, String lName,
            String dob, String address1, String address2, String postcode,
```

```
74              String phoneNum) throws ParseException {
75          position = findSolicitor(emps, empName); //find the Solicitor
76          cases.add(new Case(title, type, desc, emps.get(position)));
77          addClient(clients, fName, lName, dob, address1, address2, postcode, phoneNum, cases);
78          JOptionPane.showMessageDialog(null, "Case successfully added\nRef Num: " + clients.get(clients.size()-1).getCASE().getCASE_REF_NUM());
79      } //end method addCase
80
81
82      /**
83       * Method to add new Client to Client List.
84       * @param clients the List of Clients to be added to.
85       * @param fName the Client's first name.
86       * @param lName the Client's last name.
87       * @param dob the Client's date of birth.
88       * @param address1 the Client's address first line.
89       * @param address2 the Client's address second line.
90       * @param postcode the Client's postcode.
91       * @param phoneNum the Client's phone number.
92       * @param cases the List the Client's Case is contained in.
93       * @throws ParseException thrown if the DOB field cannot be parsed.
94       */
95      private void addClient(ArrayList<Client> clients, String fName, String lName,
96              String dob, String address1, String address2, String postcode,
97              String phoneNum, ArrayList<Case> cases) throws ParseException {
98          clients.add(new Client(fName, lName, dob, address1, address2, postcode, phoneNum, cases.get(cases.size()-1)));
99      } //end method addClient
100
101
102     /**
103      * Method to control routine of adding a new and valid Case to the List.
104      * @param caseTitleTxtField Case Title input.
105      * @param caseTypeCBox Case Type selection.
106      * @param caseDescTxtField Case Description input.
107      * @param solCBox Solicitor selection.
108      * @param fNameTxtField Client first name input.
109      * @param lNameTxtField Client last name input.
110      * @param dobTxtField Client date of birth input.
111      * @param addressFLineTxtField Client address line 1 input.
112      * @param addressSLineTxtField Client address line 2 input.
113      * @param postcodeTxtField Client postcode input.
114      * @param phoneTxtField Client phone number input.
115      * @param employees the List of employees from which a responsible Solicitor is selected and assigned to a Case.
116      * @param clients the List to add a Client object to.
117      * @param cases the List to add the new Case object to.
118      * @return true if Case is added to List; false otherwise.
119      * @throws ParseException thrown if the DOB field cannot be parsed.
120      */
121     public boolean addNewCase(JTextField caseTitleTxtField, JComboBox caseTypeCBox, JTextArea caseDescTxtField, JComboBox solCBox,
122             JTextField fNameTxtField, JTextField lNameTxtField, JTextField dobTxtField,
123             JTextField addressFLineTxtField, JTextField addressSLineTxtField,
124             JTextField postcodeTxtField, JTextField phoneTxtField,
125             ArrayList<Employee> employees, ArrayList<Client> clients, ArrayList<Case> cases) throws ParseException {
126         //Variables for a Case object
127         String title = getValidCaseTitle(caseTitleTxtField);
```

```
128         CaseType type = (CaseType) caseTypeCBox.getSelectedItem();
129         String desc=caseDescTxtField.getText();
130         String solicitor = (String) solCBox.getSelectedItem();
131         int position = 0;
132         //Variables for a Client object
133         String fName = getValidFName(fNameTxtField);
134         String lName = getValidLName(lNameTxtField);
135         String dob = getValidDob(dobTxtField);
136         String address1="", address2="";
137         if(validateAddress(addressFLineTxtField.getText(), addressSLineTxtField.getText())) {
138             address1 = addressFLineTxtField.getText();
139             address2 = addressSLineTxtField.getText();
140         } else {
141             invalidFields = invalidFields+2;
142             commonElements.showError(error11, "Invalid Address");
143             commonElements.outlineRed(addressFLineTxtField);
144             commonElements.outlineRed(addressSLineTxtField);
145         } //end if else
146         String postcode = getValidPostcode(postcodeTxtField);
147         String phoneNum = getValidPhone(phoneTxtField);
148 
149         if(invalidFields == 0) { //data is ready for Controller to construct new object
150             addCase(position, employees, solicitor, cases,
151                                 title, type, desc, clients, fName, lName,
152                                 dob, address1, address2, postcode, phoneNum);
153             return true;
154         } else {
155             invalidFields = 0; //reset invalidFields ready for next attempt
156             return false;
157         } //end if else
158     } //end method addNewCase
159 
160 
161     /**
162      * Method to check for empty fields
163      * @param caseTitleTxtField Case Title input
164      * @param caseDescTxtField Case Description input
165      * @param fNameTxtField Client first name input
166      * @param lNameTxtField Client last name input
167      * @param dobTxtField Client date of birth input
168      * @param addressFLineTxtField Client address first line input
169      * @param addressSLineTxtField Client address second line input
170      * @param postcodeTxtField Client postcode input
171      * @param phoneTxtField Client phone number input
172      * @return emptyFields the number of empty fields
173      */
174     public int checkEmpty(JTextField caseTitleTxtField, JTextArea caseDescTxtField, JTextField fNameTxtField,
175             JTextField lNameTxtField, JTextField dobTxtField, JTextField addressFLineTxtField,
176             JTextField addressSLineTxtField, JTextField postcodeTxtField, JTextField phoneTxtField) {
177         int numEmptyFields = 0;
178         if(caseTitleTxtField.getText().equals("Enter case title") || isEmpty(caseTitleTxtField.getText())) {
179             commonElements.outlineRed(caseTitleTxtField);
180             numEmptyFields++;
181         }
```

```
182         if(caseDescTxtField.getText().equals("Enter description here") || isEmpty(caseDescTxtField.getText())) {
183             caseDescTxtField.setBorder(new LineBorder(Color.RED, 2));
184             numEmptyFields++;
185         }
186         if(fNameTxtField.getText().equals("Enter first name") || isEmpty(fNameTxtField.getText())) {
187             commonElements.outlineRed(fNameTxtField);
188             numEmptyFields++;
189         }
190         if(lNameTxtField.getText().equals("Enter last name") || isEmpty(lNameTxtField.getText())) {
191             commonElements.outlineRed(lNameTxtField);
192             numEmptyFields++;
193         }
194         if(dobTxtField.getText().equals("DD/MM/YYYY") || isEmpty(dobTxtField.getText())) {
195             commonElements.outlineRed(dobTxtField);
196             numEmptyFields++;
197         }
198         if(addressFLineTxtField.getText().equals("Address line 1") || isEmpty(addressFLineTxtField.getText())) {
199             commonElements.outlineRed(addressFLineTxtField);
200             numEmptyFields++;
201         }
202         if(addressSLineTxtField.getText().equals("Address line 2") || isEmpty(addressSLineTxtField.getText())) {
203             commonElements.outlineRed(addressSLineTxtField);
204             numEmptyFields++;
205         }
206         if(postcodeTxtField.getText().equals("Enter postcode") || isEmpty(postcodeTxtField.getText())) {
207             commonElements.outlineRed(postcodeTxtField);
208             numEmptyFields++;
209         }
210         if(phoneTxtField.getText().equals("Enter phone no.") || isEmpty(phoneTxtField.getText())) {
211             commonElements.outlineRed(phoneTxtField);
212             numEmptyFields++;
213         }
214         return numEmptyFields;
215     } //end method checkEmpty
216 
217 
218     /**
219      * Method to find element in List.
220      * @param employees the List to be searched.
221      * @param solicitor the name selected by the user.
222      * @return position the position at which selected element is found in List.
223      */
224     private int findSolicitor(ArrayList<Employee> employees, String solicitor) {
225         int position = 0;
226         ListIterator<Employee> lIterator = employees.listIterator();
227         while(lIterator.hasNext())
228             if(lIterator.next().getFullName().equals(solicitor))
229                 position = lIterator.previousIndex();
230         return position;
231     } //end method findSolicitor
232 
233 
234     /**
235      * Method to retrieve all Solicitor names.
```

```
236      * Names stored in one-dimensional array for use in JComboBox.
237      * @param employees the List of Employee objects to be iterated over.
238      * @return empNames a 1D array of Employee object names.
239      */
240     public String[] getNames(ArrayList<Employee> employees) {
241         String[] empNames = new String [employees.size()];
242         for(int i = 0; i < empNames.length; i++) {
243             empNames[i] = employees.get(i).getFullName();
244         } //endfor
245         return empNames;
246     } //end method getNames
247 
248 
249     /**
250      * Method to send Case Title data for validation.
251      * Called in addNewCase method.
252      * If input is invalid, increments invalidFields by 1.
253      * @return title the valid Case Title
254      */
255     private String getValidCaseTitle(JTextField caseTitleTxtField) {
256         String title = "";
257         if(validateCaseTitle(caseTitleTxtField.getText())) {
258             title = caseTitleTxtField.getText();
259         } else {
260             invalidFields++;
261             commonElements.showError(error4, "Invalid");
262             commonElements.outlineRed(caseTitleTxtField);
263         } //end if else
264         return title;
265     } //end method getCase
266 
267 
268     /**
269      * Method to get a valid date of birth value.
270      * Checks date in valid format and contains valid value.
271      * @return date a valid date of birth.
272      */
273     private String getValidDob(JTextField dobTxtField) {
274         String dob="";
275         if(validateDateFormat(dobTxtField.getText())) { //if the format is valid
276             if(validateDate(dobTxtField.getText())) { //if the value is valid
277                 dob = dobTxtField.getText();
278             } else { //invalid date value
279                 invalidFields++;
280                 commonElements.showError(error7, "DOB Invalid");
281                 commonElements.outlineRed(dobTxtField);
282             } //end if else
283         } else { //invalid date format
284             invalidFields++;
285             commonElements.showError(error6, "DOB Format Invalid");
286             commonElements.outlineRed(dobTxtField);
287         } //end if else
288         return dob;
289     } //end method getValidDob
```

```
290
291
292     /**
293      * Method to receive valid first name value
294      * @return fName a valid first name
295      */
296     private String getValidFName(JTextField fNameTxtField) {
297         String fName="";
298         if(validateName(fNameTxtField.getText())) {
299             fName = fNameTxtField.getText();
300         } else {
301             invalidFields++;
302             commonElements.showError(error5, "Invalid Name");
303             commonElements.outlineRed(fNameTxtField);
304         } //end if else
305         return fName;
306     } //end method getValidFName
307
308
309     /**
310      * Method to receive valid last name value
311      * @return fName (String) a valid last name
312      */
313     private String getValidLName(JTextField lNameTxtField) {
314         String lName="";
315         if(validateName(lNameTxtField.getText())) {
316             lName = lNameTxtField.getText();
317         } else {
318             invalidFields++;
319             commonElements.showError(error5, "Invalid Name");
320             commonElements.outlineRed(lNameTxtField);
321         } //end if else
322         return lName;
323     } //end method getValidLName
324
325
326     /**
327      * Method to receive a valid phone number value
328      * @param phoneNumTxtField Client phone number input
329      * @return phoneNum (String) a valid phone number value
330      */
331     private String getValidPhone(JTextField phoneNumTxtField) {
332         String phoneNum = "";
333         if(validatePhoneNum(phoneNumTxtField.getText())) {
334             phoneNum = phoneNumTxtField.getText();
335         } else {
336             invalidFields++;
337             commonElements.showError(error9, "Invalid Phone Number");
338             commonElements.outlineRed(phoneNumTxtField);
339         } //end if else
340         return phoneNum;
341     } //end method getValidPhone
342
343
```

```
344     /**
345      * Method to receive a valid postcode value
346      * @param postcodeTxtField Client postcode input
347      * @return postcode a valid postcode value
348      */
349     private String getValidPostcode(JTextField postcodeTxtField) {
350         String postcode = "";
351         if(validatePostcode(postcodeTxtField.getText())) {
352             postcode = postcodeTxtField.getText();
353             //make sure postcode in correct format ready for storage
354             if(!postcode.contains(" ")) {
355                 // 1. split the postcode
356                 String outwardCode = postcode.substring(0, postcode.length()-3).toUpperCase();
357                 String inwardCode = postcode.substring(postcode.length()-3).toUpperCase();
358                 // 2. update postcode to include space
359                 postcode = outwardCode + " " + inwardCode;
360             } //endif
361         } else {
362             invalidFields++;
363             commonElements.showError(error8, "Invalid Postcode");
364             commonElements.outlineRed(postcodeTxtField);
365         } //end if else
366         return postcode;
367     } //end method getValidPostcode
368
369
370     /**
371      * Method to check for an empty field.
372      * @param text the String value to be checked.
373      * @return true if String is empty; otherwise false.
374      */
375     private boolean isEmpty(String text) {
376         return (text.equals("") || text==null) ? true : false;
377     }  //end method isEmpty
378
379
380     /**
381      * Method to validate an address against regular expression.
382      * @param address1 the String value that makes up part of an address.
383      * @param address2 the String value that completes the address.
384      * @return true if both Strings conform to regex; false otherwise.
385      */
386     private boolean validateAddress(String address1, String address2) {
387         return (address1.matches("^[\\n\\r\\x20-\\x7E]+$") && address2.matches("^[\\n\\r\\x20-\\x7E]+$")) ? true : false;
388     } //end method valid
389
390
391     /**
392      * Method to validate a Case title against a regular expression.
393      * @param title the String value to be validated.
394      * @return true if String matches the regular expression; false otherwise.
395      */
396     private boolean validateCaseTitle(String title) {
397         return title.matches("^[A-Za-z ]+ v [A-Za-z ]+$");
```

```
398     } //end method validateCaseTitle
399
400
401     /**
402      * Method to make use of Pattern and Matcher classes to validate a date.
403      * @param dob the date to be validated.
404      * @return  true if date value is valid; false otherwise.
405      */
406     private boolean validateDate(String dob) {
407         boolean valid = false;
408         Pattern datePattern = Pattern.compile("(0[1-9]|[12][0-9]|[3][01])/(0[1-9]|1[012])/(\\d{4})");
409         Matcher dateMatcher = datePattern.matcher(dob);
410         if(dateMatcher.matches()) { //if the format is valid
411             String year = dateMatcher.group(3); //segment group 3 to access YYYY
412             if(year.charAt(0) == '1' || year.charAt(0) == '2') { //check first digit is 1 or 2
413                 DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
414                 dateFormat.setLenient(false);  //dob must match SimpleDate pattern exactly
415                 try {
416                     dateFormat.parse(dob); //parse the String value
417                     valid = true; //all validation checks have been passed, the data is acceptable
418                 } //end try
419                 catch (ParseException p) {
420                     commonElements.showError(error7, "Invalid"); //invalid values
421                 } //end try catch
422             } //endif
423         } //else the format is invalid
424         else
425             commonElements.showError(error6, "Invalid"); //invalid format
426         //endif
427         return valid;
428     } //end method validateDate
429
430
431     /**
432      * Method to validate the format of a date.
433      * @param dob the String value to evaluate.
434      * @return true if date is in acceptable format; otherwise false.
435      */
436     private boolean validateDateFormat(String dob) {
437         return dob.matches("(0[1-9]|[12][0-9]|[3][01])/(0[1-9]|1[012])/(\\d{4})");
438     } //end method validateDateFormat
439
440
441     /**
442      * Method to validate a name against regular expression.
443      * @param name the name to be validated.
444      * @return true if name is acceptable; false otherwise.
445      */
446     private boolean validateName(String name) {
447         return name.matches("^[A-Z][a-zA-Z]{1,25}$");
448     } //end method validateName
449
450
451     /**
```

```
452      * Method to validate a UK phone number against regular expression.
453      * @param phoneNum the phone number to be validated.
454      * @return true if phone number is acceptable; false otherwise.
455      */
456     private boolean validatePhoneNum(String phoneNum) {
457         return (phoneNum.matches("^0[0-9]{10}") || phoneNum.replaceAll(" ", "").matches("^0[0-9]{10}")) ? true : false;
458     } //end method validatePhoneNum
459
460
461     /**
462      * Method to validate a UK postcode against regular expression.
463      * Regular expression abstracted from GOV.UK.
464      * Can be found at the following address:
465      *
    https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/488478/Bulk_Data_Transfer_-_additional_validation_
    valid_from_12_November_2015.pdf
466      * @param postcode the postcode to be validated.
467      * @return true if postcode is acceptable; false otherwise.
468      */
469     private boolean validatePostcode(String postcode) {
470         //make sure postcode in correct format ready for validating
471         if(!postcode.contains(" ")) {
472             // 1. split the postcode
473             String outwardCode = postcode.substring(0, postcode.length()-3);
474             String inwardCode = postcode.substring(postcode.length()-3);
475             // 2. update postcode to include space
476             postcode = outwardCode + " " + inwardCode;
477         } //endif
478         if (postcode.matches("^([Gg][Ii][Rr] 0[Aa]{2})|((([A-Za-z][0-9]{1,2})|(([A-Za-z][A-Ha-hJ-Yj-y][0-9]{1,2})|(([A-Za-z][0-9][A-Za-z])|([A-Za-
    z][A-Ha-hJ-Yj-y][0-9]?[A-Za-z])))) [0-9][A-Za-z]{2})$"))
479             return true;
480         return false;
481     }// end method validatePostcode
482 } //end class NewCaseController
```