Assignment 4 (v1.2)

Programming in Java

2014–15

1 Introduction

Launching a start-up company is difficult. Apart from creating a great product, you must talk to a lot of people to get them interested in the project: clients, venture capital, government agencies... it is really difficult to keep track of everybody!

When is your next meeting? Who will you meet? What did they say the last time you talked to them?

The purpose of this assignment it writing a program to keep track of contacts and meetings. The application will keep track of contacts, past and future meetings, etc. When the application is closed, all data must be stored in a text file called "*contacts.txt*". This file must be read at startup to recover all data introduced in a former session (the format of the file if up to you: you can use XML, comma-separated values (CSV), or any other format).

2 Implementation details

The program should implement the interfaces provided below. Classes that implement an interface should follow the convention of having the same name as the interface with the "Impl" suffix unless there is more than one. For example, if there is only one class that implements `ContactManager`, it must be called `ContactManagerImpl`. Be careful with this, as it may interfere with automatic tools that will analyse your code.

In order to complete this piece of coursework, you must follow the Test Driven Development (TDD) methodology. That is, you must first write the interfaces (most of them are already provided below), then write the tests, then write the implementation. Your commit history should clearly show that you wrote the tests before the implementation and not after, and that you added new functionality in small cycles of interface–test–implementation and not big chunks of untested code in one go.

2.1 ContactManager

```
import java.util.Calendar;
import java.util.List;
import java.util.Set;

/**
 * A class to manage your contacts and meetings.
 */
public interface ContactManager {
  /**
   * Add a new meeting to be held in the future.
   *
   * @param contacts a list of contacts that will participate in the meeting
   * @param date the date on which the meeting will take place
   * @return the ID for the meeting
   * @throws IllegalArgumentException if the meeting is set for a time in the past,
   *     of if any contact is unknown / non-existent
   */
  int addFutureMeeting(Set<Contact> contacts, Calendar date);

  /**
```

```
 * Returns the PAST meeting with the requested ID, or null if it there is none.
 *
 * @param id the ID for the meeting
 * @return the meeting with the requested ID, or null if it there is none.
 * @throws IllegalArgumentException if there is a meeting with that ID happening in the future
 */
PastMeeting getPastMeeting(int id);

/**
 * Returns the FUTURE meeting with the requested ID, or null if there is none.
 *
 * @param id the ID for the meeting
 * @return the meeting with the requested ID, or null if it there is none.
 * @throws IllegalArgumentException if there is a meeting with that ID happening in the past
 */
FutureMeeting getFutureMeeting(int id);

/**
 * Returns the meeting with the requested ID, or null if it there is none.
 *
 * @param id the ID for the meeting
 * @return the meeting with the requested ID, or null if it there is none.
 */
Meeting getMeeting(int id);

/**
 * Returns the list of future meetings scheduled with this contact.
 *
 * If there are none, the returned list will be empty. Otherwise,
 * the list will be chronologically sorted and will not contain any
 * duplicates.
 *
 * @param contact one of the user's contacts
 * @return the list of future meeting(s) scheduled with this contact (maybe empty).
 * @throws IllegalArgumentException if the contact does not exist
 */
List<Meeting> getFutureMeetingList(Contact contact);

/**
 * Returns the list of meetings that are scheduled for, or that took
 * place on, the specified date
 *
 * If there are none, the returned list will be empty. Otherwise,
 * the list will be chronologically sorted and will not contain any
 * duplicates.
 *
 * @param date the date
 * @return the list of meetings
 */
List<Meeting> getFutureMeetingList(Calendar date);

/**
 * Returns the list of past meetings in which this contact has participated.
 *
 * If there are none, the returned list will be empty. Otherwise,
 * the list will be chronologically sorted and will not contain any
```

```
 * duplicates.
 *
 * @param contact one of the user's contacts
 * @return the list of future meeting(s) scheduled with this contact (maybe empty).
 * @throws IllegalArgumentException if the contact does not exist
 */
List<PastMeeting> getPastMeetingList(Contact contact);

/**
 * Create a new record for a meeting that took place in the past.
 *
 * @param contacts a list of participants
 * @param date the date on which the meeting took place
 * @param text messages to be added about the meeting.
 * @throws IllegalArgumentException if the list of contacts is
 *      empty, or any of the contacts does not exist
 * @throws NullPointerException if any of the arguments is null
 */
void addNewPastMeeting(Set<Contact> contacts, Calendar date, String text);

/**
 * Add notes to a meeting.
 *
 * This method is used when a future meeting takes place, and is
 * then converted to a past meeting (with notes).
 *
 * It can be also used to add notes to a past meeting at a later date.
 *
 * @param id the ID of the meeting
 * @param text messages to be added about the meeting.
 * @throws IllegalArgumentException if the meeting does not exist
 * @throws IllegalStateException if the meeting is set for a date in the future
 * @throws NullPointerException if the notes are null
 */
void addMeetingNotes(int id, String text);

/**
 * Create a new contact with the specified name and notes.
 *
 * @param name the name of the contact.
 * @param notes notes to be added about the contact.
 * @throws NullPointerException if the name or the notes are null
 */
void addNewContact(String name, String notes);

/**
 * Returns a list containing the contacts that correspond to the IDs.
 *
 * @param ids an arbitrary number of contact IDs
 * @return a list containing the contacts that correspond to the IDs.
 * @throws IllegalArgumentException if any of the IDs does not correspond to a real contact
 */
Set<Contact> getContacts(int... ids);

/**
 * Returns a list with the contacts whose name contains that string.
```

```
   *
   * @param name the string to search for
   * @return a list with the contacts whose name contains that string.
   * @throws NullPointerException if the parameter is null
   */
  Set<Contact> getContacts(String name);

  /**
   * Save all data to disk.
   *
   * This method must be executed when the program is
   * closed and when/if the user requests it.
   */
  void flush();
}
```

2.2 Contact

```
/**
 * A contact is a person we are making business with or may do in the future.
 *
 * Contacts have an ID (unique), a name (probably unique, but maybe
 * not), and notes that the user may want to save about them.
 */
public interface Contact {
  /**
   * Returns the ID of the contact.
   *
   * @return the ID of the contact.
   */
  int getId();

  /**
   * Returns the name of the contact.
   *
   * @return the name of the contact.
   */
  String getName();

  /**
   * Returns our notes about the contact, if any.
   *
   * If we have not written anything about the contact, the empty
   * string is returned.
   *
   * @return a string with notes about the contact, maybe empty.
   */
  String getNotes();

  /**
   * Add notes about the contact.
   *
   * @param note the notes to be added
   */
  void addNotes(String note);
}
```

2.3 Meeting

```
import java.util.Calendar;
import java.util.Set;

/**
* A class to represent meetings
*
* Meetings have unique IDs, scheduled date and a list of participating contacts
*/
public interface Meeting {
  /**
   * Returns the id of the meeting.
   *
   * @return the id of the meeting.
   */
  int getId();

  /**
   * Return the date of the meeting.
   *
   * @return the date of the meeting.
   */
  Calendar getDate();

  /**
   * Return the details of people that attended the meeting.
   *
   * The list contains a minimum of one contact (if there were
   * just two people: the user and the contact) and may contain an
   * arbitraty number of them.
   *
   * @return the details of people that attended the meeting.
   */
  Set<Contact> getContacts();
}
```

2.4 PastMeeting

```
/**
 * A meeting that was held in the past.
 *
 * It includes your notes about what happened and what was agreed.
 */
public interface PastMeeting extends Meeting {
  /**
   * Returns the notes from the meeting.
   *
   * If there are no notes, the empty string is returned.
   *
   * @return the notes from the meeting.
   */
  String getNotes();
}
```

2.5 FutureMeeting

```
/**
 * A meeting to be held in the future
 */
public interface FutureMeeting extends Meeting{
    // No methods here, this is just a naming interface
    // (i.e. only necessary for type checking and/or downcasting)
}
```

3 Submission and grading

The assignment must be pushed to a project called `ContactManager` on your GitHub account. It will be automatically cloned on the date specified on the Moodle site at 23h59.59 London time. You are encouraged to leave everything ready well in advance —both the programming and pushing it to your GitHub account— to avoid last-minute problems (e.g. GitHub may be down on that weekend for maintenance). If the code is not available at the GitHub account by the deadline, the assignment will not be marked.

The assignment will be graded according to its compliance with the provided specification; the simplicity, clarity, and generality of the code (including succint but illustrative comments and JavaDoc); and the compliance with good practices of version control (e.g. commiting often and in small pieces, use of descriptive commit messages, committing only source code and not binary or class files).

Regardless of the times you choose to *push* your changes to GitHub, you should *commit* early and often. In case of suspected plagiarism, your version control history will be used as additional evidence to judge the case. It is in your best interest to commit very often (and to use adequate commit messages) to make it clear that the process of creation is entirely your own.