Test Cases of Telegram

Team Foobar: Yijia Zhang, Xiling Li, Bingfei Zhang

In this report we will first discuss the role of test cases in Telegram and what test cases we have developed for Telegram.

As a unique repo, this time Telegram surprises us again by not containing test cases in the repo. This might be due to the fact that the test cases are for private usages, but since we have no means of contacting the developers we would have to live with that. Trying to cover the system with a comprehensive test suite would be virtually an impossible mission, but writing structural-oriented test cases for certain parts of the implementation can certainly help us deepen our understanding toward the system as we have to read through the code to do so. Moreover, we may be able to reveal bugs that are not detectable when the app is being run.

For a system as gigantic and complex as Telegram, most of the components are hard for us to test without mocking or stubbing excessive dependencies. However, this also gave us a hint of why the project contained so many huge files, which sometimes seems to be breaking the single-responsibility principle. With so many interdependent modules, it might be the optimal design choice to put certain classes and functionalities together so that developers can better search and reference them. However, there are still many important components of the system that are not designed to be conveniently testable. The usages of singletons, for example, are mostly optimal in terms of design and safety. But they make testing harder by limiting the access to instances and having global static fields.

The component we first decided to test is ChatObject, which is mainly responsible for determining and retrieving user's right to perform chat-related actions such as adding administrators and sending or editing messages. Before looking carefully at the class, we pictured the user action administration system as a simple one: administrators being omnipotent with regular users being… just users. But looking at the implementation detail, we discovered that rights to many actions can be individually configurable and it is NOT true that administrators have unlimited power. Actions can be "Bannable", and not all of the actions are defaulted to be empowered to administrators.

```
private static boolean isBannableAction(int action)
    switch (action) {
        case ACTION_PIN:
        case ACTION_CHANGE_INFO:
        case ACTION_INVITE:
        case ACTION_SEND:
        case ACTION_SEND_MEDIA:
        case ACTION_SEND_STICKERS:
        case ACTION_EMBED_LINKS:
        case ACTION_SEND_POLLS:
        case ACTION_VIEW:
            return true;
    }
    return false;
}
```

Bannable actions

```
private static boolean isAdminAction(int action)
    switch (action) {
        case ACTION_PIN:
        case ACTION_CHANGE_INFO:
        case ACTION_INVITE:
        case ACTION_ADD_ADMINS:
        case ACTION_POST:
        case ACTION_EDIT_MESSAGES:
        case ACTION_DELETE_MESSAGES:
        case ACTION_BLOCK_USERS:
            return true;
    }
    return false;
}
```

Admins do not have all rights

The difficulty of writing tests for this class is that, with so many “categories” and conditions of rights, it is hard to effectively partition the inputs. Divide-and-conquering, we had to write 10 test cases just to test two methods as exhaustively as we could. While doing so we, again, saw the importance of utilizing test method names, as they can be very efficient in telling how test inputs are partitioned.

```
@Test
public void canUserDoBannedAction(){
    chat.banned_rights.change_info = true;
    chat.banned_rights.pin_messages = true;

    assertFalse(chatObject.canUserDoAction(chat, ChatObject.ACTION_PIN));
    assertFalse(chatObject.canUserDoAction(chat, ChatObject.ACTION_ADD_ADMINS));
}

@Test
public void canUserDoBannableNonAdminAction(){
    assertTrue(chatObject.canUserDoAction(chat, ChatObject.ACTION_SEND));
    assertTrue(chatObject.canUserDoAction(chat, ChatObject.ACTION_VIEW));
    assertTrue(chatObject.canUserDoAction(chat, ChatObject.ACTION_SEND_POLLS));
}

@Test
public void canUserDoBannableActionIntypesOfChats(){
    TLRPC.TL_chat_layer92 chat_layer92 = new TLRPC.TL_chat_layer92();
    assertTrue(chatObject.canUserDoAction(chat_layer92, ChatObject.ACTION_SEND));   //non-admin action, bannable
    assertTrue(chatObject.canUserDoAction(chat_layer92, ChatObject.ACTION_CHANGE_INFO)); //admin action, bannable

    TLRPC.TL_chat_old tl_chat_old = new TLRPC.TL_chat_old();
    assertTrue(chatObject.canUserDoAction(tl_chat_old, ChatObject.ACTION_SEND));
    assertTrue(chatObject.canUserDoAction(chat_layer92, ChatObject.ACTION_CHANGE_INFO));

}
```

Having informative long method names is helpful

The other component we intended to test is the method which is responsible for the different sorting options when viewing contacts, as there are two options: by name or by last seen time. The method in source code that is directly related to this function is setSortType() in ContactsAdapter.

```
public void setSortType(int value) {
    sortType = value;
    if (sortType == 2) {
        if (onlineContacts == null) {
            onlineContacts = new ArrayList<>(ContactsController.getInstance(currentAccount).contacts);
            int selfId = UserConfig.getInstance(currentAccount).clientUserId;
            for (int a = 0, N = onlineContacts.size(); a < N; a++) {
                if (onlineContacts.get(a).user_id == selfId) {
                    onlineContacts.remove(a);
                    break;
                }
            }
        }
        sortOnlineContacts();
    } else {
        notifyDataSetChanged();
    }
}
```

The method first sorts the contacts as needed and then notify RecyclerView of the data change. We thought this would be a rather straightforward test as the partitioning and the data involved for sorting is clear. However, during constructing test cases, the dependencies are more complicated than we thought it would be. For instance, it is related to a certain user's remote database, where not only ContactsController but also MessageController is involved. We tried using Mockito and PowerMock but currently didn't manage to develop a proper test suite for the method.

As there is no test cases in the original repo, we decided not to make a pull request to it for the test cases, but instead committed to our own forked repo, where the new test

cases and the intended trying can be found at https://github.com/xilingl1/Telegram in the newest two commits.